NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PATH CALCULATION AND PACKET TRANSLATION FOR UAV SURVEILLANCE IN SUPPORT OF WIRELESS SENSOR NETWORKS

by

Stephen Schall

September 2006

Thesis Advisor:	Gurminder Singh
Thesis Co-Advisor:	Ravi Vaidyanathan

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2006	Master's Thesis

4. TITLE AND SUBTITLE Path Calculation and Packet Translation for UAV Surveillance in Support of Wireless Sensor Networks	5. FUNDING NUMBERS
6. AUTHOR(S) Stephen Schall	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Wireless Sensor Networks (WSNs) are a relatively new technology with many potential applications, including military and homeland security surveillance operations. Accurate classification of WSN contacts has been attempted using various sensor combinations over the past few years, yet video and photographic imagery remain the only choices for attaining context specific contact classification. While cameras have been successfully installed within some WSNs, there are serious limitations to this solution. Most stemming from the scarce power resources, immobility, and small form factor common among conventional WSN nodes. An efficient, low cost answer to this problem involves the use of unmanned aerial vehicles (UAVs) to acquire imagery of WSN contacts. For this system to scale to the wide expanses that WSNs deploy over, UAV contact surveillance operations must be controlled autonomously. The objective of this thesis is to research and implement an autonomous UAV—WSN system, where an optimized two-dimensional flight plan is produced in response to WSN contact detection. Flight plans autonomously guide the UAV on a course to either an estimated interception point with the WSN contact or to the instigated WSN cluster, depending upon user input. The event driven application produced in this study functions in the periphery of the Kestrel Autopilot System, communicating flight plans to the UAV through properly crafted Kestrel packets.

14. SUBJECT TERMS Wireless Sensor Network, Contact Interception, Mote, MMALV, Unicorn, Kestrel Autopilot System, Procerus, UAV Path-planning	15. NUMBER OF PAGES 191
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

PATH CALCULATION AND PACKET TRANSLATION FOR UAV SURVEILLANCE IN SUPPORT OF WIRELESS SENSOR NETWORKS

Stephen A. Schall
Ensign, United States Navy
B.S., United States Naval Academy, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Stephen Schall

Approved by: Gurminder Singh
Thesis Advisor

Ravi Vaidyanathan
Thesis Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Wireless Sensor Networks (WSNs) are a relatively new technology with many potential applications, including military and homeland security surveillance operations. Accurate classification of WSN contacts has been attempted using various sensor combinations over the past few years, yet video and photographic imagery remain the only choices for attaining context specific contact classification. While cameras have been successfully installed within some WSNs, there are serious limitations to this solution. Most stemming from the scarce power resources, immobility, and small form factor common among conventional WSN nodes. An efficient, low cost answer to this problem involves the use of unmanned aerial vehicles (UAVs) to acquire imagery of WSN contacts. For this system to scale to the wide expanses that WSNs deploy over, UAV contact surveillance operations must be controlled autonomously. The objective of this thesis is to research and implement an autonomous UAV—WSN system, where an optimized two-dimensional flight plan is produced in response to WSN contact detection. Flight plans autonomously guide the UAV on a course to either an estimated interception point with the WSN contact or to the instigated WSN cluster, depending upon user input. The event driven application produced in this study functions in the periphery of the Kestrel Autopilot System, communicating flight plans to the UAV through properly crafted Kestrel packets.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS AND ACRONYMS

VCDI	Virtual Cockpit Development Interface
INT	Integer (4 Bytes)
UINT	Unsigned Integer (2 Bytes)
CHAR	Character (1 Byte)
UCHAR	Unsigned Character (1 Byte)
OTAv1	Object Tracking Application version 1
COTS	Commercial Off the Shelf
WSN	Wireless Sensor Network
PIR	Passive Infrared
RF	Radio Frequency
DoS	Denial of Service
CH	Cluster Head
BS	Base Station
UNPF	Unified Network Protocol Framework
VC	Virtual Cockpit 2.1
PCPTAv1	Path Calculation and Packet Translation Application version 1
RC	Remote Control
GUI	Graphical User Interface
HUD	Heads-up-display
MMALV	Morphing Micro Air-Land Vehicle
MAV	Micro Air Vehicle
CMOS	Complementary Metal-Oxide Semiconductor

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to first thank my mother and father. The guidance, patience and loving support you have shown me through my life resonates in everything I do. Dalaine, thank you for your tireless support and the occasional distraction during the many long hours spent working on this project.

The efforts of my thesis advisors in shaping the direction and quality of this thesis were invaluable. I would like to express my most sincere gratitude to Professors Gurminder Singh and Ravi Vaidyanathan for their counsel and assistance in refining this study.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Wireless sensor networks (WSNs) have the potential to revolutionize the way our military and law enforcement combat today's most pressing issues such as the drug trade, illegal immigration, terrorism and human trafficking. They are a viable solution to the growing need for manpower to watch expansive foreign and domestic borders and other hotbeds for the aforementioned illegal activities. Once deployed, a wireless sensor network will automatically sense and report anything that moves through it while collecting a variety of data that can be used to classify each contact. A properly orchestrated wireless sensor network would make it possible for one man to stand watch over an area that would otherwise require the assistance of many more. The utilization of this technology will not only improve the efficiency of US and Coalition forces, it will save resources and lives. With the accurate, real-time intelligence supplied, proportional responses can be mounted against any intrusion, ensuring threats are always met with an adequate reactionary force.

Information superiority is vital to the success of US and Coalition Forces against an increasingly asymmetrical enemy. The Global War on Terrorism will probably never truly end; however, it will be intelligence, not overwhelming firepower that brings about periodic victories. The ability to deploy various sensors along sensitive borders and in other areas of interest would significantly improve the United State's intelligence collection capacity. WSNs have the potential to become the eyes and ears of the United States military, dissolving the need for a persistent, predictable physical presence during surveillance or defensive operations.

B. PROBLEM

The level of response, if any, that a WSN contact warrants is difficult to evaluate. WSNs can send an alert upon the detection of a contact detailing its speed and a reasonably accurate guess as to what the contact is. The most advanced of which can distinguish between a man and a machine with the help of a magnetometer. The contact's sound resonance can also be used to assist in further narrowing down its

classification. But even sensor networks outfitted with this level of sophistication fall prey to the same shortcomings as their predecessors. Sensor networks cannot tell what flag is painted on the side of a passing truck, what activity the contact is engaged in, if there is a warrant out for a contact's arrest, or any other context specific information. It is unrealistic to expect a reconnaissance team to investigate each contact, especially in a high traffic area.

Logic would suggest attaching a camera, actuated in the presence of a contact, to the WSN motes themselves, and this was in fact the topic of several master's theses. While it is a worthwhile addition, this implementation is not a sufficient standalone solution. There is no guarantee a stationary mote camera would be in a position to acquire useful footage. Sensor network motes are designed to stay out of sight, meaning most have a small stature and are easily obscured by vegetation. Additionally, barring the case where a sensor network is deployed along a road, many of its contacts will not behave in an ideal manner. Some contacts may pass through the center of the network, but others may blow through the outside corner. If a stationary camera is able to acquire footage in this situation, it is unlikely that it would provide an informative angle. Finally, a mote camera would be limited to its surrounding area, leaving the network administrator to assess the situation with only a small piece of the overall picture.

C. SOLUTION

Unmanned aerial vehicles (UAVs) when integrated with a WSN can be used to capture video of a WSN contact, which would then be viewed by a human operator to determine the appropriate level of response. They have the flexibility to move with a contact, even after it has left the reach of its associated WSN. They can provide a bird's eye view of the surrounding area, allowing the network administrator to ascertain a contact situation in its entirety. After all, one sensor network contact may be the first soldier in a long convoy. UAVs can be manually controlled or programmed to position themselves in the most advantageous vantage point to acquire quality footage of a contact. The latter of which roughly describes one aspect of a fully autonomous system, where an interesting contact would automatically trigger the launch of a UAV for further inspection. This automation is key to the scalability potential of the WSN—UAV system and to the reduction in the number of administrators required to watch a specific area.

D. SCOPE/ORGANIZATION

This thesis first investigates the three main components that together form the WSN—UAV system. These are the WSN, the UAV, and the hardware/software that enables the two to communicate. The coverage of Chapter II extends to all three. First, an overview of current WSN technology will be conducted, as an understanding of this technology is essential for any reader to appreciate the ideas presented in this thesis, as well as the motivation driving this endeavor. Accordingly, prior work investigating the viability of outfitting WSN motes with cameras is included in this section. Second, the Unicorn and MMALV UAV platforms utilized in this study are profiled. The third background section explores the Kestrel Autopilot System provided by Procerus, the company responsible for the Unicorn in its entirety and the most of the MMALV internals. The Kestrel Autopilot System is a collection of hardware and software that enables the UAV to be autonomously controlled from the ground.

The main focus of this thesis is in the automation of the UAV from the point of WSN contact data receipt, to UAV—contact interception. This research produced an application capable of guiding any UAV utilizing the Kestrel Autopilot System from its current location to the estimated position of a WSN contact, along an optimized path. The design and implementation of this software is detailed in Chapter III, and the accuracy of its calculations are evaluated in Chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. INTRODUCTION

An examination of the component hardware and software necessary to implement an autonomous UAV—WSN system is detailed in this chapter. We will begin with an overview of WSN technology and its developmental state, and follow with a look at the work completed by others in the realm of WSN contact classification. Then the UAV platforms used during this study are featured as well as the autonomous autopilot package used to control them. It is hoped that the information provided here will aid the reader in both their appreciation and analysis of the autonomous UAV—WSN integration project introduced in Chapter III.

B. WIRELESS SENSOR NETWORKS

Wireless sensor networks (WSN) represent the fusing of two well known technologies into one functional system. Both wireless networks and environmental sensors have received much attention in the way of research and constructive development. However, the evolution of WSNs is a relatively new phenomenon that has opened the doors to many exciting advances in the realm of pervasive computing.

1. Introduction to Wireless Sensor Networks (WSNs)

A WSN is an interconnected ad-hoc mesh system of small, low-cost sensing nodes that send observed sensory data to a specific collection node over radio frequency (RF) communication. These sensor nodes are commonly composed of a "processing unit with limited computational power and limited memory, sensors (including specific conditioning circuitry), a communication device, and a power source in the form of a battery" (Wikipedia: "Wireless Sensor Network," 2006).

The purpose of these networks is to autonomously collect various data about an operational environment. Theoretically, hundreds or even thousands of these low cost nodes could be deployed in an operational area of interest. Upon deployment, sensor nodes automatically collaborate and form a meshed network supported by RF communication, then begin collecting sensor data without any assistance or input from the user. This self-organizing characteristic is intrinsic to all ad-hoc devices. The

meshed nature of inter-node communications yields a self-healing wireless network, ensuring node connectivity is as adaptable as hardware and power limitations will allow.

Sensor data from each sensor node must be sent to a collection node for it to be utilized. This collection point usually houses user applications that parse and display the data in a viewable form. There are many networking architectures and routing schemes that support the movement of data from one network node to another; however, special consideration must be taken in the case of WSNs because the method employed can weigh heavily on each node's limited battery life.

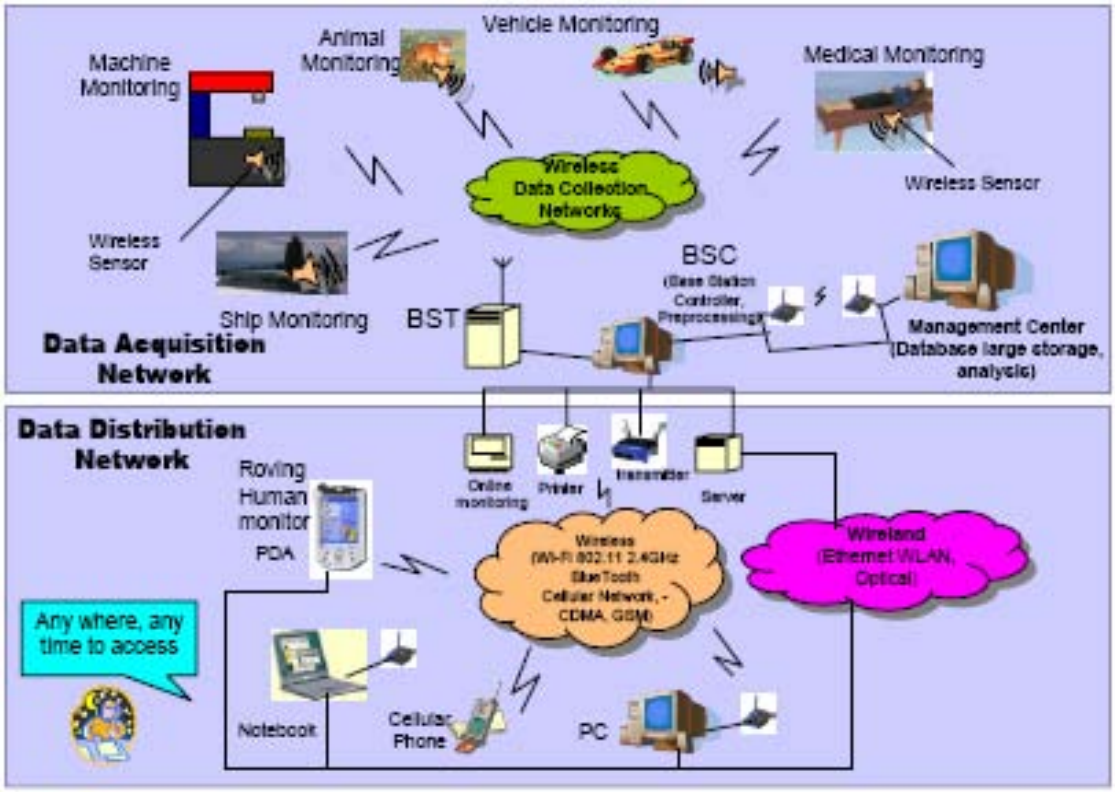


Figure 1. Wireless Sensor Network (From: Lewis, 2004)

2. Applications

WSNs can be constructively applied to almost any situation where additional information about one's environment would be helpful. It is difficult to imagine any activity where a wider awareness of one's surroundings would not improve the safety or effectiveness of the endeavor. Thus the potential applications of this technology are numerous and varied. It is perhaps easiest to organize a discussion as to the relevancy of WSNs in the manner dictated by Culler, Estrin and Srivastava. The multitude of both

potential and realized uses for WSNs can be roughly categorized into those that observe "space, things or the interactions of things" (Culler, David, Estrin and Srivastava, 2006).

The activities mentioned here in no way constitute a conclusive list of each category's occupants. That said, some of the applications falling into the first group are environmental, biocomplexity (Estrin, Srivastava and Sayeed, 2002) and seismic monitoring, military surveillance, international border/treaty enforcement, indoor climate control, and precision agriculture (Culler, David, Estrin and Srivastava, 2006). The second group includes inventory tracking, centralized control over home appliances, health monitoring, "condition-based equipment maintenance" (Culler, David, Estrin and Srivastava, 2006), and structural analysis (Bharathidasan, Archana and Ponduru, 2006). Some of the most profound WSN applications observe the interactions of various entities within a system. These include ecosystem monitoring, asset tracking, disaster recovery, contaminant transport, educational tools, interactive toys, and ubiquitous computing support.

3. Wireless Sensor Network Characteristics

a. Wireless Sensor Nodes

Wireless sensor nodes have a number of defining features. Network nodes are small, lightweight, low cost, and are expected to function for extended periods of time under the constrictions of very limited energy resources. In some WSNs, more than one type of node is used to perform the various tasks that the network requires. There are a number of workable variations that can be made to a basic heterogeneous WSN. But the idea in its most general form is that there are at least two functional subsets into which a network's nodes fall; whether these subsets are distinguished by differences in a node's hardware, software, middleware or physical appearance is not of consequence. For example, consider a system whereby one type of node collects sensory data and the other's sole responsibility is to create a data sink into which sensory nodes report their observations. In another common variant of the heterogeneous variety, all nodes within the network collect sensory data, and the only distinction between them exists in a node's

logical stature within the network's data flow hierarchy. The alternative to utilizing several different node types is of course a homogenous system, where each node performs the same role.

b. Network Architectures

Given a WSN with heterogeneous nodes, data aggregation at one point in the network promotes comprehensive sensory updates and centralized control. Cluster architectures were developed decades ago to control the flow of data within wireless networks. This architecture type is a good fit for WSNs, as it supports efficient communications and scalability (Al-Karaki, "Handbook of Sensor Networks"). Cluster architectures are logically built upon a tree structure, with all data flowing to one or more root nodes. More specifically, nodes positioned within the same general locality pass data to a cluster head (CH), which forwards the data to a base station (BS) for processing. Although this allows sensory data from large portions or the entire network to be collected, processed and displayed to the user at one point, the base station as well as the other nodes residing at a similar hierarchical level will bear the brunt of the network traffic and communications overhead.

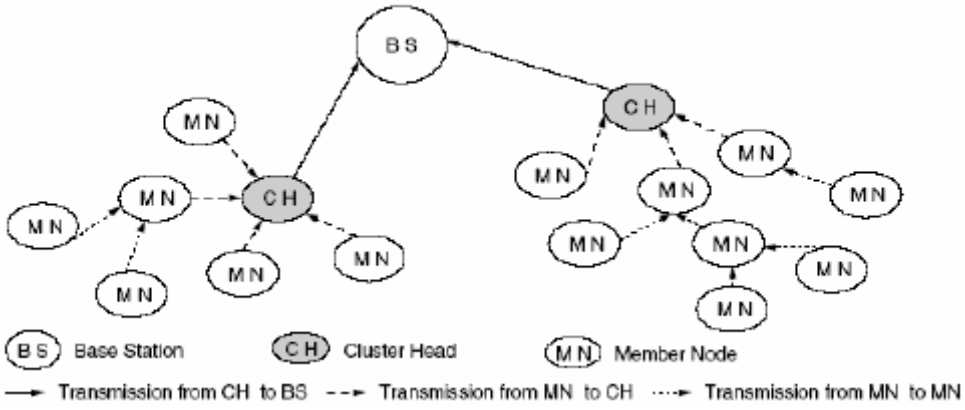


Figure 2. Multihop clustering architecture (Yarvis and Ye, 2005)

Layered architectures are able to exploit the relative low cost of homogeneous nodes. Nodes are grouped into layers based on their hop count from the network BS. Protocols that function according to this architecture attempt to minimize the RF range of the nodes composing each layer, balancing energy consumption against network interconnectivity. The Unified Network Protocol Framework (UNPF) is a set of

such protocols. UNPF protocols perform a series of three steps to caste the network into layers and govern inter-node communications. In the first step, commonly referred to as the network initialization, the BS transmits a unique discovery message over a network control channel. Nodes close enough to receive this message form the first layer. These nodes then transmit their own control channel discovery messages, and the receiving nodes become layer two. This continues until there is no response to a layer's discovery messages. Sensory data can then begin to flow from one layer to the next toward the network BS. UNPF protocols initiate the node discovery and layering process periodically to account for failing or displaced nodes.

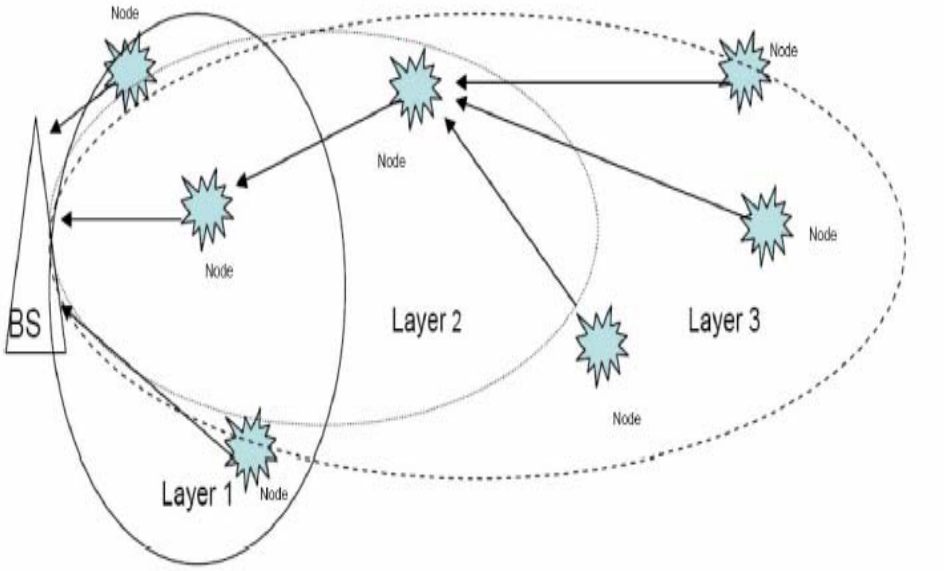


Figure 3. Layered Network Architecture (From: Murthy and Manoj, 2004)

4. Power Management Considerations

WSN nodes must function under extreme power consumption frugality due to the physical characteristics WSN nodes are expected to exhibit and the desire for nodes to function in a self-sufficient manner for as long as possible. They must remain small to avoid being seen and therefore cannot house a large power source. They must be lightweight to support rapid deployment scenarios and as a result cannot incur the added weight of a heavy power source. Generally, cutting edge portable power sources are too expensive to incorporate into a device that is designed to be low-cost and/or disposable.

In many of the applications for which they were designed, circumstances make it either impractical or unsafe to change the batteries of a node once deployed in its area of operation. These demands place considerable restriction on an already limited field of available portable power technologies. Regardless of a node's portable power source, if its power supply is not renewable, careful power management will always play a significant roll in the node's longevity.

Figure 4. WSN node RF Transmission/Receive/Process Energy Consumption (From: Estrin, Srivastava and Sayeed, 2002)

a. Routing Protocols

There are many routing protocols that have been applied to WSNs, and they can be categorized into one of two groups: those that take power management into consideration and those that do not. Most members of the latter group utilize some variant of a technique called flooding, in which nodes broadcast received packets to each of their neighbors, regardless of whether or not a destination node has already received the packet (Bharathidasan, Archana and Ponduru, 2006). This routing scheme is inefficient because resources are wasted sending a packet to nodes that already have it. Typically, a simplistic approach such as this will also lack the functionality to adapt its routing scheme to the network's dynamic energy landscape. There are obviously many routing protocols that make no effort to avoid gross energy consumption inefficiencies, but the following focuses on a few of the sophisticated approaches that do.

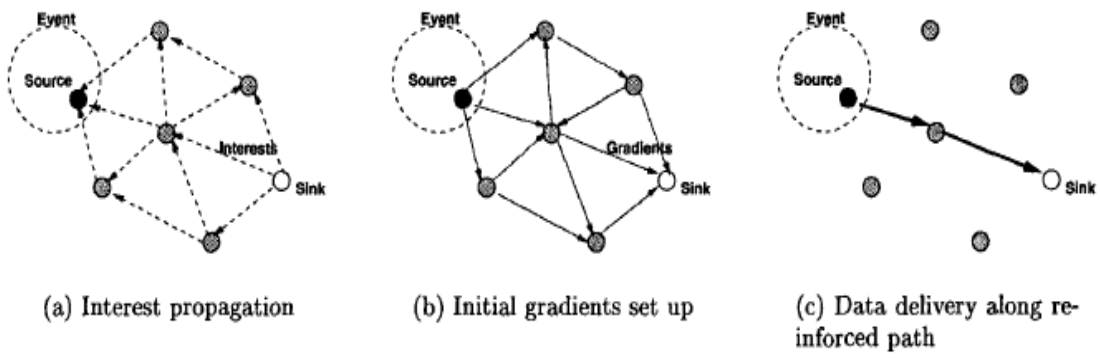
Sensor Protocols for Information via Negotiation (SPIN) is a class of WSN routing protocols that includes SPIN-PP, SPIN-EC, SPIN-BC, and SPIN-RL (Bharathidasan, Archana and Ponduru, 2006). These protocols employ two power-saving methods. First, SPIN nodes negotiate communications before transferring data. Using

imbedded information descriptors (meta-data) detailing the contents of the intended transfer, a SPIN node broadcasts an "ADV message" to its neighbors before any data is sent to prevent wasting resources sending the data to a node that already has it (Kulik, Heinzelman and Balakrishnan, 2002). Nodes that do not yet have the data send a "REQ message" to the source of the ADV message (Kulik, Heinzelman and Balakrishnan, 2002). The initiating node then sends a "DATA message" containing the actual data and a meta-data header that aids the destination in associating the DATA message with its REQ (Bharathidasan, Archana and Ponduru, 2006). The second power saving technique SPIN protocols employ is they force nodes to "poll a resource manager" before communicating or processing received packets to ensure they have enough power to perform the operation and continue functioning regularly.

In Shah and Rabaey's 2002 paper, "Energy Aware Routing for Low Energy Ad Hoc Sensor Networks," they describe "a destination initiated reactive protocol…that instead of maintaining one optimal path, maintains a set of good paths that are chosen from by means of a probability which depends on how low the energy consumption of each path is" (Bharathidasan, Archana and Ponduru, 2006). Some "energy efficient" protocols find the path of least energy consumption and continually utilize that route until the nodes in it are depleted. Failure of nodes along a particular path could leave a section of the WSN alienated from the sensor data collection point. Shah and Rabaey's method forces the WSN to utilize its combined energy reserves, allowing node resources to deplete collectively. Nodes in this scheme use localized packet flooding to set up routing (interest) tables (Shah and Rabaey, 2002). Several paths are created to each destination. Paths that have high energy costs relative to the other options are discarded and a proportion is assigned to each remaining path based on the energy cost required to use it. When data is sent across the network, nodes select a destination path probabilistically from the choices found in their interest tables (Shah and Rabaey, 2002).

Directed diffusion is a data-centric routing scheme in which nodes request data by broadcasting an 'interest' to their neighbors (Intanagonwiwat, Govindan and Estrin, 2000). Gradients are established within the network, detailing the direction data associated with a specific interest should be sent upon receipt. These gradients point

back to the node that originally broadcasted the interest request (see Figure 5). When nodes within the WSN receive data matching the interest request, nodes begin sending the data down many different paths, in the direction of the established gradients. "The sensor network reinforces one, or a small number of these paths" (Intanagonwiwat, Govindan and Estrin, 2000). This routing paradigm improves WSN energy economy because the decision as to which path is reinforced can incorporate energy considerations. Additionally, once interest paths are reinforced nodes recognize where to pass different data, saving them from costly flooding or destination finding operations.



(a) Interest propagation (b) Initial gradients set up (c) Data delivery along reinforced path

Figure 5. Directed Diffusion Routing (From: Intanagonwiwat, Govindan and Estrin, 2000)

b. Redundancy vs. Power Economy

A delicate balance must be maintained between the redundancy of the WSN mesh and the energy draw of its nodes. Stronger RF transmissions give nodes a longer communications range, which allows them to mesh with a larger number of their counterparts. This adds to the number of routing paths available to each node, enabling them to route around failing neighbors or adjust to a change in topology. However, the increased redundancy of the mesh comes at a high energy cost. RF communications draw substantially more power as their range is increased.

c. Sensor Power Saving Strategies

The energy resources required to support a node's sensors can be minimized through selective use of passive and active sensors, and by decreasing the range of active sensors. In one selective use implementation, nodes only turn on power

hungry sensors (such as GPS, cameras, sonar, etc.) when alerted to the presence of a contact by low-power sensors (such as barometers, thermometers, magnetometers, etc.). Another variant to this idea involves modifying internal nodes to leave their sensors off until a contact is detected by nodes along the WSN perimeter. The obvious drawback to this design is that perimeter nodes will exhaust their energy reserves well before those in the interior. Given that nodes are able to sense their location relative to one another and adjust their sensor settings accordingly, the WSN will shrink as each perimeter node dies. But the longevity of the system will be extended considerably.

d. Time Division Multiple Access

Time Division Multiple Access (TDMA) is a bandwidth sharing scheme that allots a specific time slot to each RF network interface. It requires nodes to have synchronized clocks. TDMA can save WSNs significant energy resources because it allows nodes to turn off their radio while it is not their turn to transmit or receive.

e. Sustainable Power

Looking to the future, outfitting each node with solar power collection cells seems to be the most promising solution to a survivable power supply. As Figure 6 shows, this technology is not tractable to indoor environments. Sustainable WSNs will only support low-power sensor types, since the usable energy will consist only of what can be replaced through solar-power collection. The small form factor nodes are expected to exhibit will not lend favorably to their collection potential, and solar-power collection is currently an inefficient energy conversion process. The present state of solar power technology will not allow for the support of most WSNs in a sustainable manner. However, solar energy can in most circumstances be counted upon to prolong the life of any outdoor WSN significantly.

	Power (Energy) Density	Source of Estimates
Batteries (Zinc-Air)	1050 -1560 mWh/cm^3 (1.4 V)	Published data from manufacturers
Batteries(Lithium ion)	300 mWh/cm^3 (3 - 4 V)	Published data from manufacturers
Solar (Outdoors)	15 mW/cm^2 - direct sun 0.15mW/cm^2 - cloudy day.	Published data and testing.
Solar (Indoor)	.006 mW/cm^2 - my desk 0.57 mW/cm^2 - 12 in. under a 60W bulb	Testing
Vibrations	0.001 - 0.1 mW/cm^3	Simulations and Testing
Acoustic Noise	3E-6 mW/cm^2 at 75 Db sound level 9.6E-4 mW/cm^2 at 100 Db sound level	Direct Calculations from Acoustic Theory
Passive Human Powered	1.8 mW (Shoe inserts >> 1 cm^2)	Published Study.
Thermal Conversion	0.0018 mW - 10 deg. C gradient	Published Study.
Nuclear Reaction	80 mW/cm^3 1E6 mWh/cm^3	Published Data.
Fuel Cells	300 - 500 mW/cm^3 ~4000 mWh/cm^3	Published Data.

With aggressive energy management, ENS might live off the environment.

Figure 6. Power source density (From: Estrin, Deborah, Mani Srivastava, and Akbar Sayeed, 2002)

5. Confidentiality, Integrity and Accessibility

The security threats to WSNs are as diverse as their applications are varied. One must consider the fact that many WSN applications necessitate deployment and operation in hostile environments. For example, military WSN security concerns include the proliferation of data to opposition forces, resource accessibility and the authenticity of gathered data. Hostile environments are not limited to the physical realm. It can be said that a wireless network is always in a hostile environment, as anyone could attempt to break into or capture packets from the network at any time. Luckily, there are some precautions that can be employed to make one's WSN more secure.

a. Confidentiality, Integrity, Authenticity

Since WSNs are ad-hoc systems, users must be concerned about where their data came from, who has seen it and whether it has been modified. With no safeguards in place, the network would happily connect to and receive sensory data from a malicious user posing as a legitimate node. This aspect of wireless security is rather straight forward. A properly orchestrated encryption scheme can guarantee confidentiality, integrity and authenticity (of sensory data). Encrypted packets will be

indecipherable to those who are not permitted to view the WSN data. Packets that are created or modified by any entity other than a valid network node will decrypt into gibberish and be discarded. In military and law enforcement applications, the encryption key must not be attainable through physical access to a node.

b. Accessibility

The wireless nature of these networks makes it nearly impossible to guarantee users persistent accessibility. Denial of service (DoS) attacks are the Achilles' heel of wireless networks. With the proper equipment, this is the easiest way to disable a WSN. All wireless networks lack the inherent (OSI model) physical layer security built into a wired network. "Malicious packets can not be prevented from reaching an access point or client as opposed to wired networks where some filtering can be employed or access to a network port can be controlled" (Egli, 2006). Wireless networks also differ from wired networks at layer two of the OSI model. The function of the data link layer is to administer the wireless protocol that mediates access to the physical layer. Above layer two, there ceases to be a distinction between wired and wireless networks.

WSN nodes do not have the power resources to transmit RF signals at a high enough decibel level to thwart close physical layer DoS radio jamming attacks. "Fortunately physical layer attacks are also difficult to execute since the power of a signal loses 6dB when doubling the distance between sender and receiver" (Egli, 2006). If a WSN is spread over a large enough distance, a physical layer DoS attack would both alert network administrators to the presence of a hostile contact and indicate the assailant's general location.

Layer 2 DoS attacks are much easier to carry out. These attacks disrupt the function of the wireless protocol, usually through some form of bogus packet injection. For example, flooding nodes with replayed routing packets will quickly deplete their limited resources and possibly cause enough congestion to bring the network to a stop. Even if nodes employ some form of encryption, receive and decrypt operations become much more costly and dangerous to the network's energy supply when multiplied over thousands or millions of spurious packets.

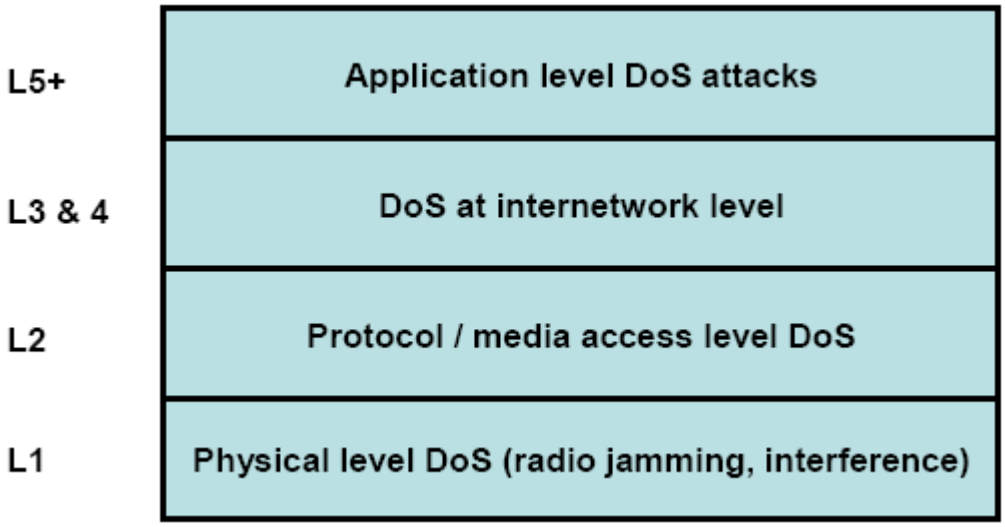


Figure 7. DoS attacks at each OSI layer (From: Egli, 2006)

6. Locality of Reference

The data collected by a WSN engaged in spatial monitoring would not be of much use without some idea as to where it came from. These applications are based on the acquisition of various measurements describing a specific locale of the WSNs operational area.

WSNs are generally composed of many nodes distributed over a large area. Incoming data is associated with a specific node through some form of addressing/header scheme, and this information is commonly coupled with the node's positional data. Whether a node's location is referenced with respect to another node or calculated independently by each node (through GPS, inertial navigation, etc.), the same effect is achieved.

7. TinyOS

Considering the limited memory, processing power and energy resources available in wireless sensor nodes, most standard operating systems are too resource hungry to sustain the levels of operative concurrency they require (TinyOS, 2006). TinyOS is an open-source, lightweight operating system designed specifically for such a role (Hill, Szewczyk, Woo, Hollar, Culler and Pister, 2006). It "is developed by a consortium led by the University of California, Berkley in cooperation with Intel Research" (Wikipedia: "TinyOS," 2006). This event-driven environment is built to

provide users with a constructive basis from which to develop small, efficient wireless sensor node applications (TinyOS, 2006). When called upon, built-in modules provide interfaces to particular data and services. Typically, this entails retrieving sensor outputs or handling device I/O (Wikipedia: "TinyOS," 2006). TinyOS also comes with a "component library," which "includes network protocols, distributed services, sensor drivers, and data acquisition tools—all of which can be used as-is or be further refined for a custom application" (TinyOS, 2006). With the aforementioned details abstracted to a particular module interface or library, users can focus on the functionality of their applications, rather than their supporting infrastructure.

C. OBJECT TRACKING APPLICATION

In 2005, a master's thesis completed by Vlasios Salatas implemented an application addressing the problem of contact detection and tracking through a sensor network. This application was simply referred to as the Object Tracking Application v1 (OTAv1).

1. Sensor Network Hardware/Software

OTAv1 operates on the periphery of a WSN developed by Crossbow Technologies. Crossbow is the self-purported "leading full-solutions supplier in the wireless sensor networking arena" (www.xbow.com). They offer a variety of products that range from full WSN solutions to supporting software and peripheral sensor network components. OTAv1 was designed to run in conjunction with the MSP410 Mote Security System.

The MSP410 System is geared toward security applications, and is maintained on the Crossbow product line for use in security applications (Crossbow Technology, 2006). Its role in supporting the OTAv1 requires it to function in a similar capacity.

Figure 8. MSP410 Mote Security System Components: 8 motes and 1 base station (From: www.xbow.com)

The MSP410 System is composed of three layers. Eight battery-powered motes sense each other's presence and connect to form a wireless ad-hoc mesh network upon deployment. MSP410 motes come from their manufacturer (Crossbow) with a pre-installed sensor suite. Each mote is armed with a 2-axis magnetometer, four passive infrared (PIR) detectors, and a dormant microphone. The mote layer, comprised of these 8 sensor motes, is responsible for collecting various sensor data in the area of operation. The flow of data in the mote layer is logically organized into a tree formation rooted at the system's BS. The Crossbow MBR410 Base Station makes up the "server layer" of the system. The server layer collects sensor and connectivity data from the mote mesh and puts it onto an RS-232 connection destined for a computer running MOTE-VIEW 1.2.

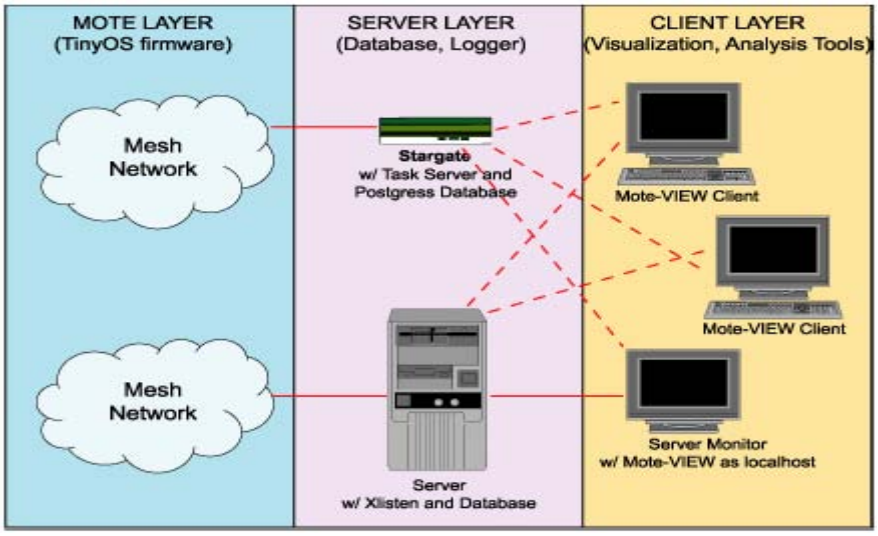


Figure 9. Layer Structure of the MSP410 Mote Security System (From: Crossbow Technology, 2006)

MOTE-VIEW was "designed to be an interface ('client layer') between a user and a deployed network of wireless sensors" (Crossbow Technology, 2006). It presents the sensor data streaming from the MSP410 base station in graphical diagrams and tables that are easily read and understood by the typical user. Figure 10 shows the layout of the "Data" tab in MOTE-VIEW. This view allows the user to monitor all the sensor values coming from the each deployed mote as well as each mote's voltage. Figure 11 presents an example of one of several graphical diagrams that MOTE-VIEW is capable of constructing to aid the user in his/her understanding of sensor data.

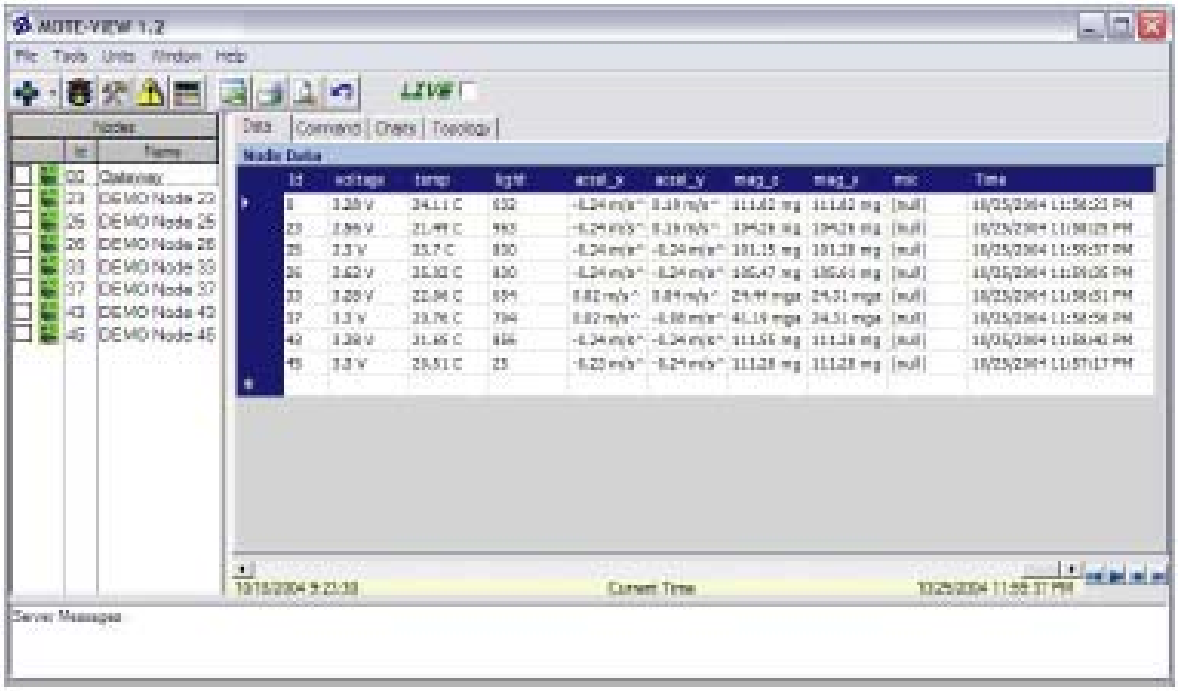


Figure 10. Sensor data from the MSP410 Mote Security System populating MOTE-VIEW (From: Crossbow Technology, 2006)

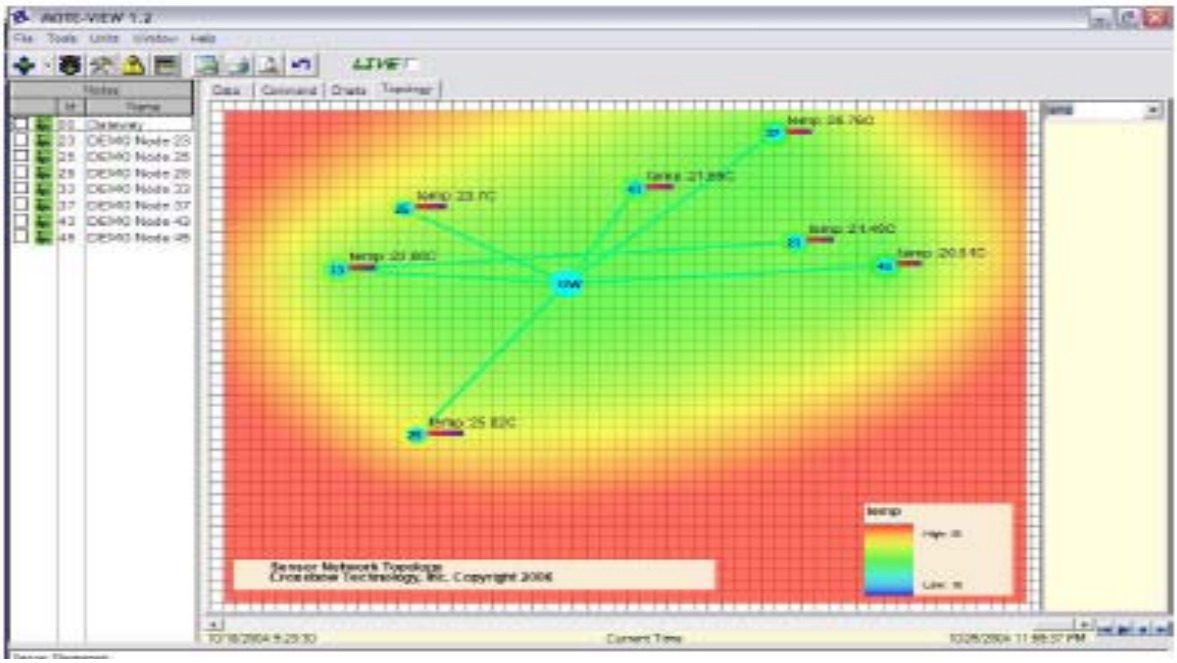


Figure 11. A topological view of the MSP410 Mote Security System depicting temperature readings in MOTE-VIEW (From: Crossbow Technology, 2006)

2. OTAv1

OTAv1 takes a continuous stream of sensor data from the WSN and analyzes this data for any indication as to the presence of a contact. If a contact is detected, OTAv1 outputs whether the contact is a human or a vehicle, the contact's speed and the contact's direction of travel through the WSN. This application is not only valuable for the descriptive data it provides on each contact, its ability to sense when a contact has entered the WSN can be used as an actuator to start other programs or devices. The software produced by Crossbow to support their WSNs can display mote sensor values in various user-friendly charts and diagrams; however, none of them contain the functionality to differentiate between mote sensor fluctuations and the presence of a contact within the WSN.

3. Object Tracking Scenarios

The focus of the Object Tracking Application project was to develop a WSN system for detecting contacts along an established path of travel. Thus, the application was limited to three scenarios. Users are responsible for correctly determining which scenario is appropriate for their particular situation. In each of these scenarios, OTAv1 expects that its associated WSN is deployed along a road, path or corridor. Although they represent simple path behavior abstractions, the applicability of the available scenarios is quite extensive, serving to ensure the program's value in both an indoor and outdoor operating environment.

The first of OTAv1's allowable scenarios is that of a WSN positioned along a straight road. The motes may be positioned either staggered on either side of the road (as in Figure 12) or in single file along one edge of the road. The deployment pattern one chooses is dependent upon the specific situational. For example, if the road is wide, it would be wise to deploy the motes on both sides of the road to avoid missing contacts on the far edge.

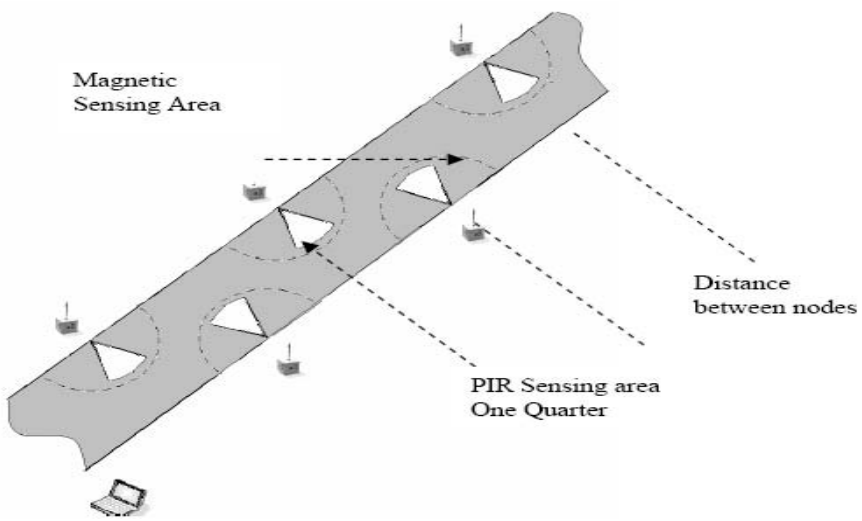


Figure 12. Topology overview of OTAv1 straight road scenario (From: Salatas, 2005)

The second and third scenarios are relatively similar to the first. They handle a T-road and 4-way intersection respectively. The mote topology is a relatively simple extension on the straight road scenario with one exception. In the T-road scenario, the data from two of the PIR sensors in the mote positioned at the intersection is incorporated. Likewise, the 4-way intersection scenario calls for the two motes positioned in its intersection to utilize two of their PIR sensors.

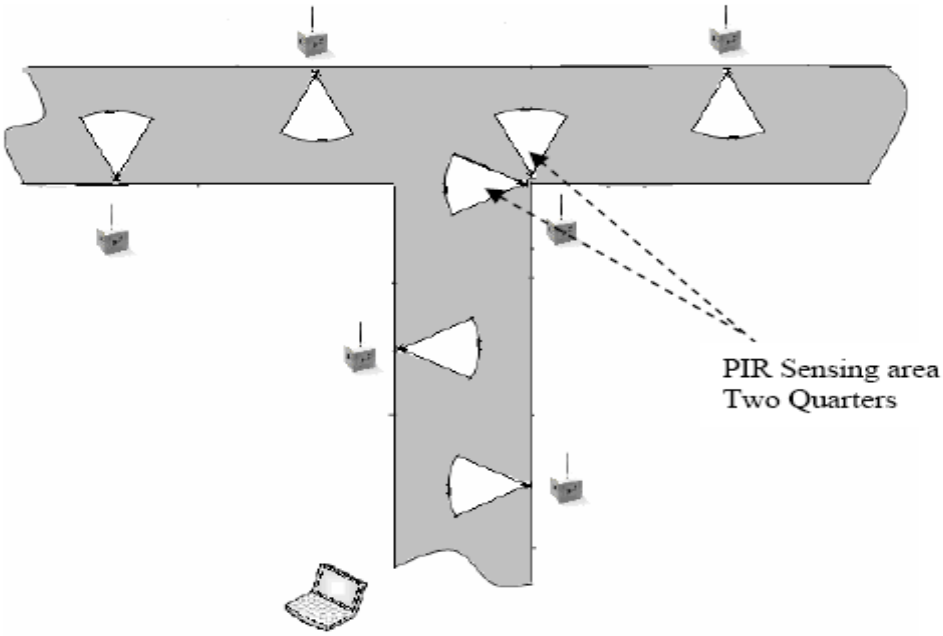


Figure 13. Topology overview of OTAv1 T-road scenario (From: Salatas, 2005)

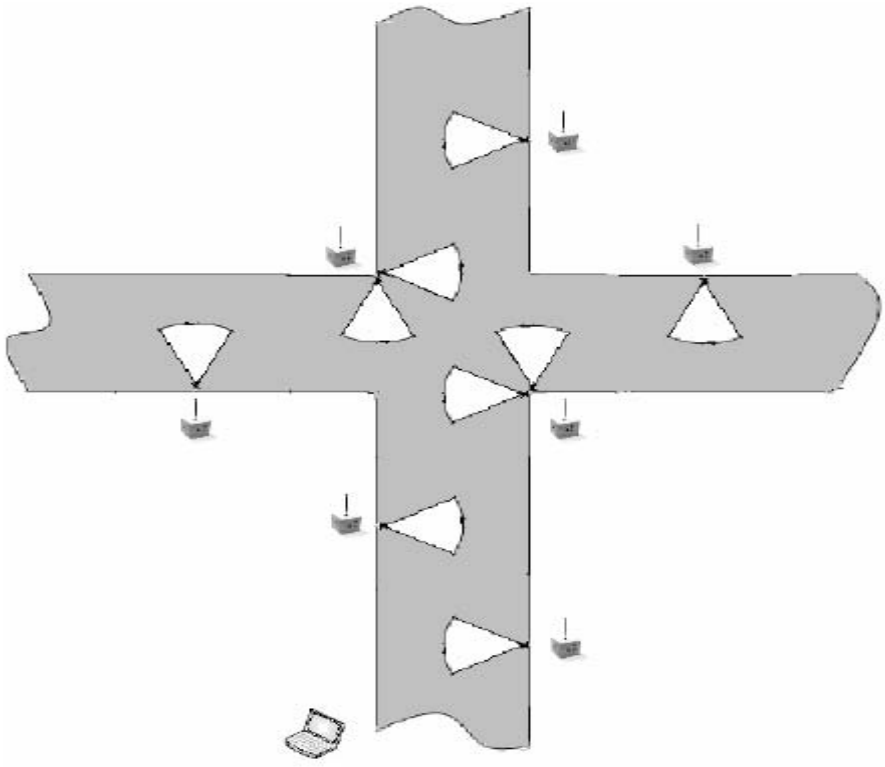


Figure 14. Topology overview of OTAv1 4-way intersection scenario (From: Salatas, 2005)

4. Contact Detection

Once the WSN is deployed in an appropriate pattern, OTAv1 can begin functioning as it was intended. OTAv1 detects contacts through a system of thresholds based on the values of each mote's magnetometer and PIR sensors. If any of the motes positioned at an extremity of a particular deployment scheme register a PIR or magnetometer reading that is greater than or equal to a predetermined value, the entrance of a contact is assumed and reported.

5. Object Identification

Once detected, contacts must be categorized as well as possible given the resources of the WSN. In this case, OTAv1 has access to magnetometer readings, PIR sensor values and all the requisite information for determining a contact's speed and direction. OTAv1 takes a logical approach towards classification in its assumption that vehicles will register relatively high magnetometer readings, while human contacts will not. Based solely on this metric, OTAv1 makes a determination as to whether a contact is a human or a vehicle. The speed with which a contact passes through the WSN is determined by tracking the time from initial contact detection to its appearance on the PIR sensor readings of motes deployed further down the road. Based on the distance between motes and the time it took for the contact to reach each successive mote, a reasonably accurate estimation of the contacts speed can be made. The contact's direction of travel is assumed to be the same as the progressive PIR sensor spikes that it causes as it passes through the WSN.

6. TRSSv3

As discussed in the Chapter I, section B, several master's theses have explored the integration of wireless cameras into sensor motes. One of the objectives of OTAv1 was to serve as the actuator for such a system. After all, the cameras can not be expected to continuously snap pictures or record video on a mote's extremely limited battery supply.

The camera integration scheme for which OTAv1 was designed to complement was developed by Brian Dixon and William Felts at the Naval Postgraduate School in 2005 and is called the Tactical Remote Sensor System (TRSSv3). The hardware components comprising TRSSv3 are one FTP server, one Globalstar Satellite phone per

mote, one 4XEM Elite2 miniPC per mote and one Creative WebCam per mote. TRSSv3 takes photos of WSN contacts using Creative WebCams upon the issuance of a contact alert by OTAv1. The cameras are triggered at the correct time through consideration of the contact speed, as estimated by OTAv1, and knowledge of their distance from the outlying WSN motes. After snapping a photo, the Creative WebCam passes its picture to a 4XEM Elite2 miniPC, which in turn forwards the picture through standard Internet routing to a Globalstar Satellite phone uplink and onto the Internet. The address of these forwarded packets is that of the FTP server, which waits for incoming pictures to host. The end result of this system is multiple hosted pictures of each contact hosted on an ftp server accessible to anyone with an Internet connection.

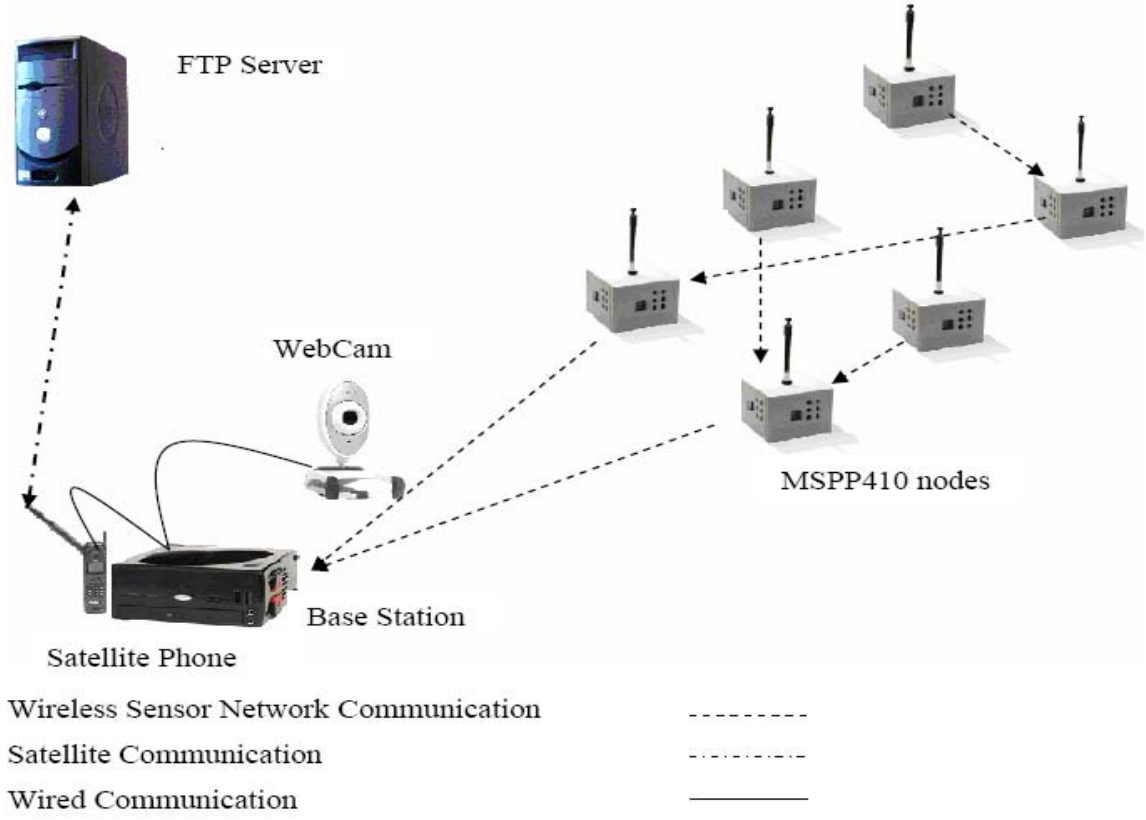


Figure 15. TRSSv3 and OTAv1 WSN System Overview (From: Salatas, 2005)

Between OTAv1 and TRSSv3, the user acquires several pictures of each contact, each contact's speed and direction, and an automated classification as to whether the contact is a human or a vehicle. While the results of the automated classification in

OTAv1 will be obvious to any human viewing the pictures from TRSSv3, the ability for the program to distinguish between people and vehicles is important if the user wishes to implement a filtering mechanism that performs a particular action depending upon the type of the contact.

D. UAV PLATFORMS

Although the Kestrel Autopilot System is compatible with a wide range of UAV platforms, only two particular aircraft were utilized during the course of this study. The Procerus Unicorn, developed by Procerus technologies, was used as a test platform for developing autonomous autopilot applications and exploring various ideas. The second platform is still under development, but will no doubt play a major role within the autonomous UAV—WSN system. The Morphing Micro Air-Land Vehicle (MMALV) was designed to take advantage of efficient locomotion models found in nature (Boria, Bachmann, Ifju, Quinn, Vaidyanathan, Perry and Wagener, "A Sensor Platform Capable of Aerial and Terrestrial Locomotion"). MMALV is the result of two merged technologies; the Micro Air Vehicle (MAV) developed by the University of Florida, and the Mini-Wheg^{TM}, developed at Case Western Reserve University. These award-winning, lightweight solutions were integrated to produce a UAV capable of aerial and terrestrial locomotion.

Figure 16. MMALV in flight (From: Boria, Bachmann, Ifju, Quinn, Vaidyanathan, Perry and Wagener, "A Sensor Platform Capable of Aerial and Terrestrial Locomotion")

1. Procerus Unicorn

a. Physical Characteristics

The Unicorn has an angled, triangular shaped chassis that measures 48 inches from wing tip to wing tip (see Figure 17). Its body is constructed with EPP foam,

making this platform particularly durable, inexpensive, and lightweight, weighing in at mere 24 oz (with prop, motor, servos and speed control) (Procerus Technologies: "UAV Test Platform," 2006). Propulsion is provided by a "brushless electric motor" that drives a rear push propeller (Procerus Technologies: "UAV Test Platform," 2006). The Unicorn's left and right elevons give it directional control, while winglets at either end "lower the lift-induced drag caused by wingtip vortices" (Wikipedia: "Winglet," 2006).

Figure 17. Procerus Unicorn, overhead view (After: Procerus Technologies: "UAV Test Platform," 2006)

With reference to Figure 17, the brown square found on the right half of the Unicorn is the base to which the GPS module is attached. The particular GPS unit utilized in this platform is the Furuno GH-81D. Shifting focus to the center of the Unicorn's frame, there are two compartments covered with Velcro flaps (see Figure 18). The larger, more centered of the two contains two lithium polymer batteries (Procerus Technologies: "UAV Test Platform," 2006). The other compartment holds the AeroComm AC4490-1000-M3 RF modem and the Kestrel Autopilot. Conveniently, these two devices connect to each other, forming one compact package that fits nicely in the allowable space. The Unicorn uses an imbedded dipole antenna to communicate with its ground station.

Figure 18. Procerus Unicorn battery (middle) and autopilot (left of center) compartments (From: Procerus Technologies: "UAV Test Platform," 2006)

b. Intended Use and Specification Overview

The Unicorn was designed for use as a test platform. The performance, low-cost and durability of the Unicorn makes it an ideal UAV from which to test a variety of onboard equipment, such as environmental sensors, various autopilot systems, modem hardware, or any other experimental payload weighing less than 16 ounces. The Unicorn's range is highly dependent upon the modem hardware and configuration used, as communications range and battery life are limiting factors in this regard. Equipped with the standard battery package and an AC4490-1000-M3 RF modem, the Unicorn has a range of "3 to 6 miles at an altitude of 400 feet" (Procerus Technologies: "UAV Test Platform," 2006). It can reach speeds of 45 MPH, with total flight endurance of about 1 to 1.5 hours (Procerus Technologies: "UAV Test Platform," 2006).

2. Morphing Micro Air-Land Vehicle

a. Intended Use

Research into deployment and reconfiguration methodologies for WSNs provided a new dimension to the MMALV project. WSNs are sometimes deployed in hostile environments, unsafe or unfit for entry. These networks must be formed by nodes capable of both deploying and relocating themselves autonomously. Outfitted with desirable WSN node hardware, the MMALV will eventually serve as a mobile WSN node (Boria, Bachmann, Ifju, Quinn, Vaidyanathan, Perry and Wagener, "A Sensor Platform Capable of Aerial and Terrestrial Locomotion"). For example, MMALV nodes composing a WSN could enter an area of particular interest piggy-backed on a larger

UAV, drop off, and fly to a programmed location. Spatial reconfiguration of the WSN would then be as simple as uploading different destinations to the MMALVs involved.

b. Wing Structure and Design

Many obstacles must be overcome to maintain flight control of "micro" UAVs, or more precisely, those falling within the Reynolds number range 10^4 and 10^6 (Mueller, 1985). In fact, the ratio of coefficient of lift to coefficient of drag drops by nearly two orders of magnitude through this range (Mueller, 1985). Additionally, the velocity of Earth's wind is comparable to the flight speed of micro UAVs, which can cause gross disparity between the forces bore by each wing. "The small mass moments of inertia of these aircraft also adversely affect the stability and control characteristics of the vehicles" (Boria, Bachmann, Ifju, Quinn, Vaidyanathan, Perry and Wagener, "A Sensor Platform Capable of Aerial and Terrestrial Locomotion").

The inspiration for solving the many stability problems resulting from micro UAV size and weight was found through observation of naturally occurring flight mechanisms, specifically the wings of birds, insects, and bats. Biological systems outperform small manmade aircraft in virtually every aspect of flight. One of the mechanisms allowing them the requisite stability to perform these maneuvers is called passive adaptive washout. The University of Florida designed the MAV with that in mind, giving it flexible bat-like wings (Ifju, Ettinger, Jenkins, Lian, Shyy and Waszak, 2002). Depending upon the airflow hitting each, MAV wings passively change shape. When a gust of wind hits a wing, the wing bends in the manner depicted by Figure 19, thus decreasing its lift efficiency. However, since the air velocity over the opposite wing is higher, it continues to develop a nearly equivalent lifting force as the left wing (Ifju, Ettinger, Jenkins, Lian, Shyy and Waszak, 2002). Both wings are capable of independently adapting to variations such as these, so flight becomes much more graceful and stabilized. The wing structure employed by the MAV has been fully integrated into the MMALV design.

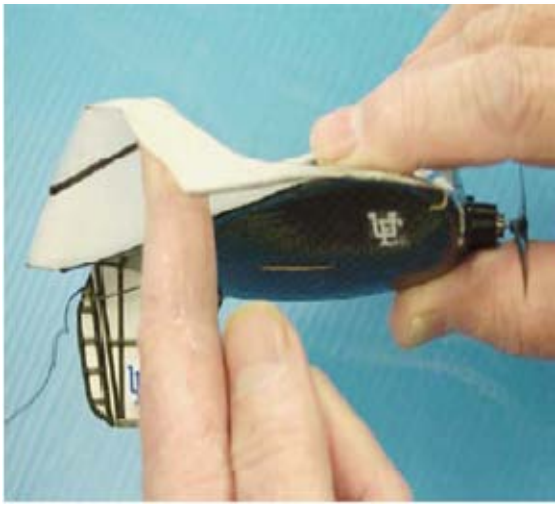


Figure 19. Demonstrating MAV wing flexibility (From: provided by Ifju Lab, University of Florida)

c. Terrestrial Locomotion

Many biologically inspired modes of terrestrial locomotion were assessed at Case Western Reserve University during the search for a system to carry the MAV airframe. The MMALV's terrestrial locomotion solution was derived from the cockroach due to its ability to traverse over both large and small obstacles with relative ease. While proceeding from one point to another with an unobstructed path, the cockroach walks with "adjacent legs 180 degrees out of phase" (Allen, Quinn, Bachmann and Ritzmann, 2003). It also exaggerates the height of each step with its front legs, "allowing it to take smaller obstacles in stride" (Allen, Quinn, Bachmann and Ritzmann, 2003). When extra stability is required to overcome larger obstacles, the cockroach brings its adjacent legs into phase (Allen, Quinn, Bachmann and Ritzmann, 2003).

Attached on either side of the MMALV's front are cockroach inspired "Mini-Whegs™ (see Figure 20). Mini-Wheg™ robots can move "with a top speed of 10 body lengths per second," drop from significant heights without damage, navigate over large obstacles, and carry a payload weighing twice their body weight (Lambrecht, Horchler and Quinn, 2005). Mini-Wheg™ technology supplies the MMALV not only with the means to efficiently crawl to a new location, but with benefits ranging from improved landing survivability to the ability to negotiate some obstacles.

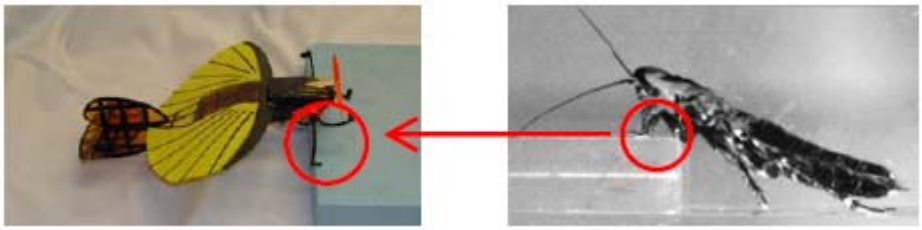


Figure 20. The inspiration for the Mini-Wheg™ design (From: Boria, Bachmann, Ifju, Quinn, Vaidyanathan, Perry and Wagener, "A Sensor Platform Capable of Aerial and Terrestrial Locomotion")

d. Hardware Layout

All of the MMALV's onboard electronics reside in the same compartment. This compartment extends from the motor (situated behind the propeller), to the beginning of the tail fin, accounting for the entire body of the aircraft. The craft's dipole antenna, used to communicate with its base station, protrudes parallel to its rudder fin from the tail of the frame.

E. KESTREL AUTOPILOT SYSTEM

The Kestrel Autopilot System is composed of four interrelated pieces: the Kestrel Autopilot, Virtual Cockpit 2.1 (VC), the Virtual Cockpit Development Interface (VCDI), and the Procerus Commbox. These components each contribute in some distinctive manner to the overall functionality of the UAV.

1. Kestrel Autopilot

The Kestrel Autopilot v2.2 is the controlling hardware and associated firmware installed in both the Unicorn and the MMALV that facilitates autonomous flight. All of the hardware hosted on the UAV is tied together and processed at the autopilot. With VC running at a ground station, users can interface with Kestrel Autopilot to receive navigational and telemetry data from a UAV, and can also update the UAV's flight plan both prior to and during flight. The Kestrel Autopilot can store flight plans consisting of up to 200 different waypoints (Procerus Technologies: "UAV Test Platform," 2006). Each UAV is assigned a unique ID number (which can be modified in VC), allowing users to operate multiple UAVs in the same area without control ambiguity.

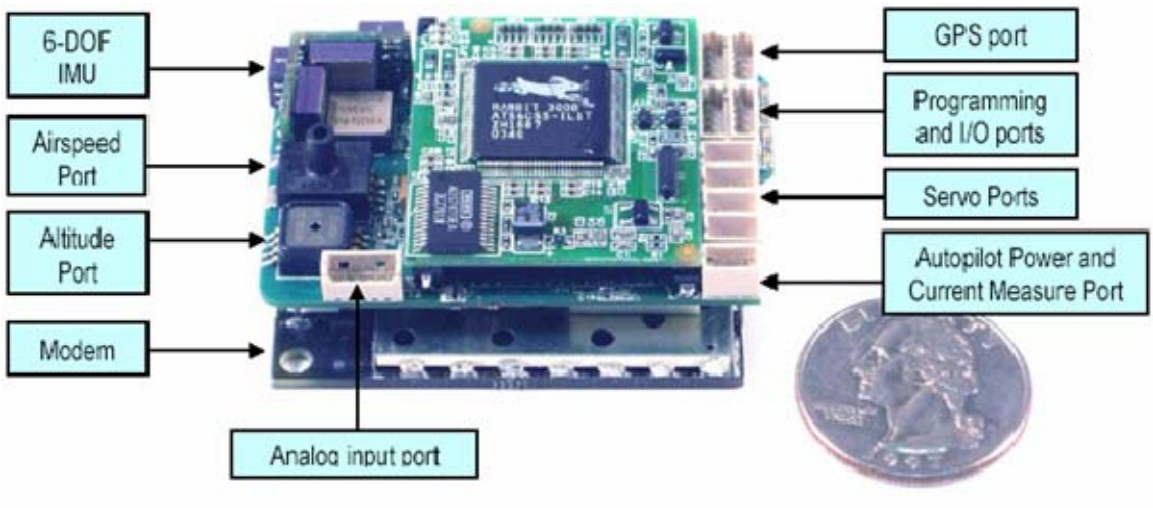


Figure 21. Kestrel Autopilot 2.2 (top) and AeroComm AC4490 RF Modem (bottom) (From: Procerus Technologies: "Kestrel Autopilot V2.22," 2006)

a. Kestrel Autopilot Hardware

The modem used by the autopilot to communicate with the ground station attaches to the bottom of the autopilot forming a two tiered circuit board (see Figure 21). The Kestrel Autopilot weighs only 16.7 grams, and requires just 1.29 in^3, making it the smallest and lightest device of its kind on the market (Procerus Technologies: "Kestrel Autopilot V2.22," 2006). It sports a Rabbit Semiconductor microprocessor, which means it operates using an 8-bit 29 MHz processor with 512K of RAM, while drawing only 0.77 Watts (Procerus Technologies: "Kestrel Autopilot V2.22," 2006).

Input Supply Voltage	-0.3V to 18V
Payload Current	500mV @ 3.3V & 5V
Operating Temperature Range	-40°C to 85°C
Storage Temperature Range	-40°C to 125°C
Maximum Absolute Pressure	400 kPa
Maximum Differential Pressure	75kPa
Humidity	5% to 95%, no condensing
Acceleration	±200 g

Table 1. Kestrel Autopilot maximum ratings (From: Procerus Technologies: "Kestrel Autopilot V2.22," 2006)

The autopilot contains an Inertial Measurement Unit (IMU), allowing it to track its current position and motion vector in real time. This data is computed through the utilization of 3-axis angular rate measurements for pitch, roll and yaw, and a

combination of GPS and 3-axis accelerometer measurements for bearing and speed (Procerus Technologies: "Kestrel Autopilot V2.22," 2006). The bearing gyro is calibrated with data from a 2-axis magnetometer when the UAV is either at a stop or traveling at a slow speed relative to the ground (Procerus Technologies: "Kestrel Autopilot V2.22," 2006). The IMU uses absolute and differential pressure sensors and GPS to obtain a measurement of its altitude.

Three temperature sensors imbedded in the autopilot enable it to automatically adjust its sensors to temperature change. This is important because temperature shifts can affect the accuracy of the IMU gyros as well as the autopilot's pressure sensors.

b. Peripheral Hardware Support

Built into the autopilot is the capacity to power peripheral hardware devices at 3.3V and 5V, both at 500mA (Procerus Technologies: "Kestrel Autopilot V2.22," 2006). It has four serial I/O ports to allow peripheral hardware devices to communicate amongst each other and to the autopilot itself. One of which is designated solely for the use of GPS. The standard serial interface of the I/O ports allows the Kestrel Autopilot to accommodate commercial off the shelf (COTS) GPS units (Procerus Technologies: "Kestrel Autopilot V2.22," 2006).

(1) Autopilot Servo Ports. The autopilot has four servo ports, which facilitate the control of the UAV's "moving parts." Servos use an electric motor to create a mechanical force that moves some piece of the UAV's body (Wikipedia: "Servomechanism," 2006). The Unicorn and MMALV have different servo arrangements due to the dissimilarity of their steering mechanisms.

(2) MMALV Servo Port Usage. The MMALV is controlled by an elevator and a rudder, accounting for two of its four servo ports. The third servo port is occupied by the throttle control. The MMALV currently requires two servo ports to operate its wegs. Since there are only four servo ports in the Kestrel Autopilot, one of two modifications must be made. It is possible to attach a servo extender board to the Kestrel Autopilot, giving it 8 additional ports. Or in the absence of an extender board, one or more of the serial I/O ports can be reprogrammed to function as servo ports.

(3) Unicorn Servo Port Usage. The Unicorn is steered with a right and left aileron, which each require servo ports. Like the MMALV, the throttle control occupies the third servo port. The Unicorn does not use the available fourth servo port.

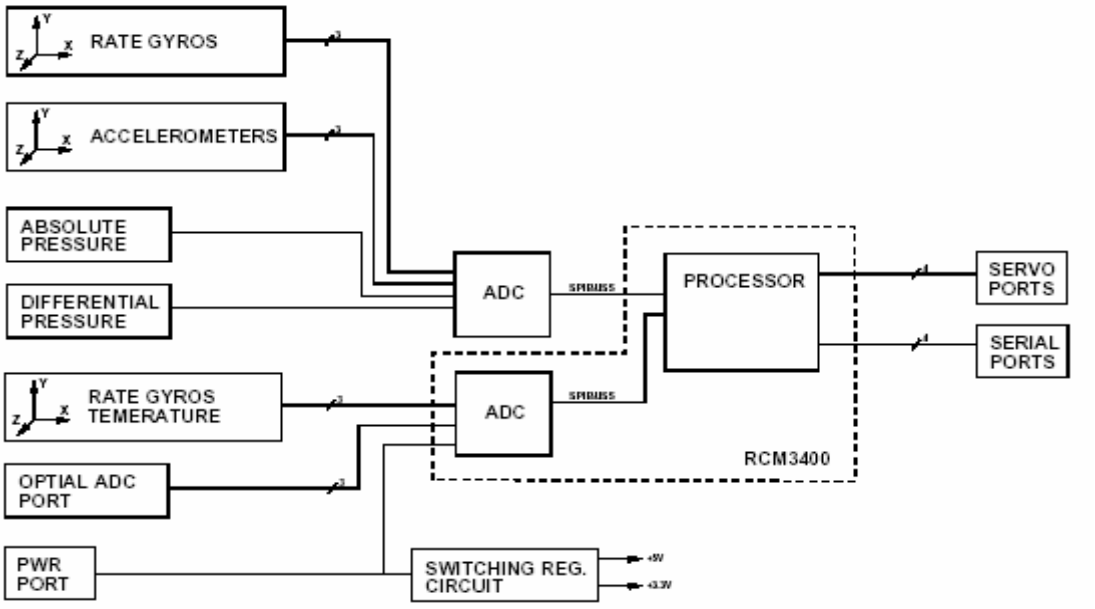


Figure 22. Kestrel Autopilot 2.2 block diagram (From: Procerus Technologies: "Kestrel Autopilot V2.22," 2006)

c. Autopilot Altitude Approximation Skew

The autopilot's reliance on pressure readings to assess its altitude has lead to a problem that was not addressed by the manufacturer. The autopilot must zero its sensors before flight to right its IMU and adjust its pressure sensors to the current reading at ground level. Based on the pressure reading at ground level, the autopilot computes its altitude. If the weather changes drastically during flight, the altitude that the autopilot associated with the ground level pressure will shift up or down depending upon the nature of the weather change. GPS is not utilized to right the pressure—altitude association because it takes the Kestrel Autopilot approximately two minutes to compute the third dimension.

2. Virtual Cockpit 2.1

Although the Kestrel Autopilot does have the capacity to function autonomously, there are still many functions that require user input or guidance, including but not limited to real-time flight plan modifications, autonomous mode switching and collision

avoidance. The user application supplying this interface is called Virtual Cockpit 2.1. VC is a Windows-based application developed by Procerus Technologies (http://www.procerusuav.com). It allows users to "configure, monitor and issue commands to the autopilot and Commbox, upload flight plans, and change waypoints, all while UAVs are in the air" (Procerus Technologies: "User Guide," 2006). It also presents all the telemetry, navigational and sensor data generated by the Kestrel Autopilot to the user through various display modules. Figure 25 illustrates the VC user interface. VC supports four control alternatives. The user can modify a UAV's flight plan through the various buttons and interactive devices found within the VC graphical user interface (GUI), through the use of the Virtual Cockpit Development Interface to send crafted Kestrel packets through VC to the Kestrel Autopilot, by controlling the UAV from VC's gamepad interface, or through the use of an RC controller. The specific gamepad with which VC was designed to function is the Logitech Dual Action Gamepad (Procerus Technologies: "User Guide," 2006).

There is an important distinction to be made between the gamepad and the RC controller that extends beyond their obvious physical differences. The gamepad is used to supplement the various modes of autonomous control offered by VC. On the other hand, once the RC controller is engaged, UAV behavior is dictated solely by the user. Users can seize control of the UAV with the RC control at any point during flight. The RC controller supersedes all other control sources.

Figure 23. RC Controller toggling Manual Mode (From: Procerus Technologies: "UAV Flight Guide," 2006)

a. Virtual Cockpit 2.1 System Requirements

According to the Kestrel Autopilot System User's Guide, any computer upon which Virtual Cockpit 2.1 will run must have at a minimum the following specifications:

- Microsoft Windows XP
- 128 MB of RAM
- 12 MB of hard drive space
- Direct X 8 compatible video card with at least 8 MB video RAM
- 700 MHz or faster CPU

b. UAV Modes

The mode buttons displayed in the upper left-hand corner of the Virtual Cockpit 2.1 GUI allow the user to select from eight varieties of autonomously controlled flight. The specific mode governing a UAV's flight can be changed at any time (Procerus Technologies: "User Guide," 2006). Beginning with the leftmost option, selecting Home mode will cause the UAV to fly to the location where the positional sensors were zeroed before flight (generally the takeoff location). Once above the intended location, the UAV will loiter in a circular holding pattern until a different mode is initiated. Selecting Takeoff mode will cause the UAV to attempt an autonomous takeoff per the parameters displayed to the user in the lower right of the VC GUI (see Figure 25). Similarly, entering into Land mode will cause the UAV to land in a manner dictated by the parameters displayed in the GUI, just below those directing autonomous takeoff. Loiter Now mode causes the UAV to enter into a circular pattern around its current location, while Rally mode causes the UAV to loiter over a user defined "approach point." The approach point is the location from which the UAV begins an autonomous landing. Although the gamepad can be used to control the UAV in any of the autonomous modes, Speed mode and Altitude mode were designed specifically to be used in conjunction with the gamepad. When the UAV switches into Speed mode, the roll angle and cruise airspeed are maintained at predefined levels, allowing the user to concentrate on altitude adjustments. This mode is helpful in assisting manual landings, since the throttle can be used to flair the vehicle before touchdown (Procerus

Technologies: "User Guide," 2006). In Altitude mode the UAV will hold a predefined altitude and roll angle (Procerus Technologies: "User Guide," 2006), leaving the user to steer and adjust the speed. This mode makes UAV piloting easier by controlling the most dangerous of the axes of movement. The gamepad's buttons take on unique meaning depending upon the selected mode, as detailed in Figure 24. The eighth and final brand of autonomous flight supported by VC is called Navigation mode. Navigation mode is fully autonomous, and so requires that the Kestrel Autopilot contain an uploaded flight plan. A UAV engaged in this mode will complete the commands in its flight plan in the order by which they were uploaded. These commands can be manipulated or replaced entirely in real-time.

Figure 24. UAV controls in Speed Mode (upper-left) and Altitude Mode (upper-right) using Logitech Dual Action Gamepad (bottom image applies to both modes), (From: Procerus Technologies: "User Guide," 2006)

c. Virtual Cockpit 2.1 Graphical User Interface

The VC GUI contains several modules, each providing the user with a variety of control options, telemetry and sensor displays. The Agent List just below the mode selection pane contains a row for every Kestrel Autopilot communicating with VC. By clicking on a UAV's address, the user can quickly switch between active agents. Only the active UAV will receive RC control packets issued by the user.

The heads-up-display (HUD), positioned under the agent list, displays the orientation of the active UAV. This pane also contains status information such as the UAV's airspeed, altitude, mode, battery voltage, and magnetometer heading.

The remaining GUI panes on the left side of the VC display are the Message Window and the Preflight Tools. The Message Window is designed to draw the user's attention to "mission critical information" (Procerus Technologies: "User Guide," 2006). The Procerus Commbox, UAV, and VC itself are all potential topics for the alerts shown in this pane. These messages are ordered and color coordinated based on their importance in the following manner: red (most important), orange, yellow, and green (least important). The Preflight Tools window has four buttons, each one initiating some aspect of the UAV's preflight regimen. The first of which is the Zero Pressure button. This button is depressed before flight while the UAV is grounded to align its sensors to ground pressure. With this information, the UAV can calculate its altitude based on the difference between the ground pressure and pressures recorded during flight. The GPS Home button causes the UAV to save its current GPS coordinates as its home location. This location should be set near the user because any severe in-flight problems will cause the UAV to fly home and loiter. The Check Sensors button must be performed at ground level before takeoff. It evaluates the sensor values returned by the Kestrel Autopilot to ensure they are "within allowable limits" (Procerus Technologies: "User Guide," 2006). The FS button opens separate window that allows the user to view and adjust the Kestrel Autopilot fail-safes. The importance of this preflight check cannot be stressed enough. Appropriate fail-safes can prevent loss or destruction of a UAV in the event of GPS signal loss, loss of communications with the ground station, low battery, or other unforeseen situations.

The most conspicuous section of the VC GUI is the Geo-referenced Map display. Users can add their own overhead map images, operational area photographs, or rely on the default grid image of the operational area. Once a satisfactory image is in place, users must provide VC with the latitude and longitude of a specified pixel within the image. The resolution of the display can be adjusted to the user's liking. When the Dwnld button is pressed, the UAV's current flight plan is sent to VC via the Procerus Commbox. This action populates the map image with an accurately scaled and positioned visual representation of the active UAV's flight plan. Users can modify any aspect of the flight plan by manipulating its visual representation. Clicking on and dragging a waypoint will reposition its associated command destination. Right-clicking on the map image allows users to add new commands to their flight plan, and commands can be deleted by selecting them and pressing the delete key. The active UAV will not adhere to any modifications made to the flight plan until the Upld button is pressed. This button directs VC to send an up-to-date flight plan to the active UAV's Kestrel Autopilot via the Procerus Commbox.

The flight plan is also displayed in numerical form in the Flight Plan pane. The Flight Plan pane supplements the Geo-referenced Map by presenting users with the same information and functionality in a different manner. Any changes made to the flight plan in the Geo-referenced Map are represented in Flight Plan pane and vice versa. Again, the Upld button must be pressed for any changes to take effect.

The only window not yet discussed is the Takeoff and Landing pane. The values found here govern the behavior of the UAV when control is passed to either autonomous Takeoff mode or Land mode. The user may modify these values at will.

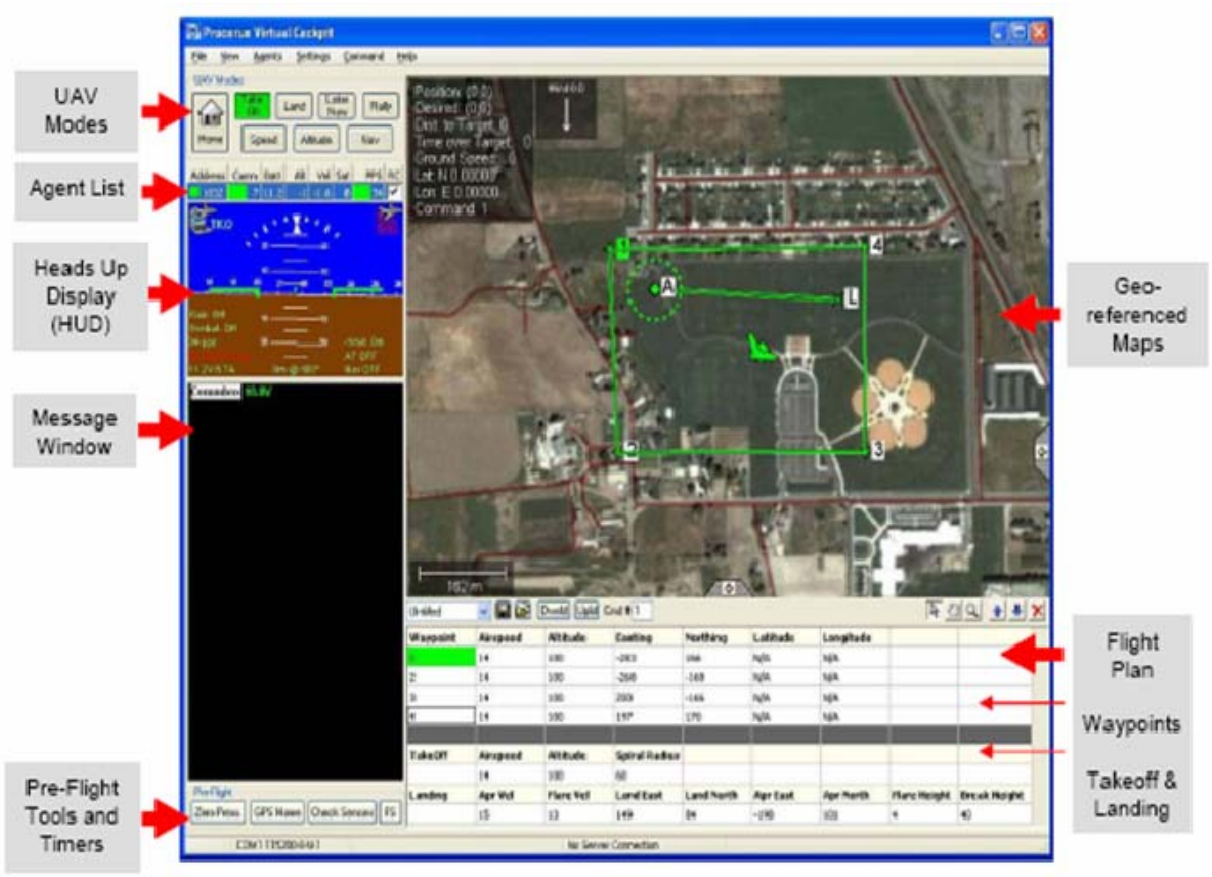


Figure 25. Virtual Cockpit 2.1 graphical user interface (From: Procerus Technologies: "User Guide," 2006)

3. Virtual Cockpit Development Interface

The Virtual Cockpit Development Interface allows users to send packets through Virtual Cockpit 2.1 to the Kestrel Autopilot, and receive packets sent from the Kestrel Autopilot via Virtual Cockpit 2.1. The VCDI is in essence a TCP/IP server that is started automatically when Virtual Cockpit 2.1 is opened. The VCDI server listens on port 5005 by default (Procerus Technologies: "Kestrel Autopilot System," 2006). As the name implies, it is intended to serve as tool for developers to build upon the functionality of Virtual Cockpit 2.1. User applications (clients) requiring a connection to the VCDI can use a loop-back IP address (127.0.0.0/8) and the VCDI listening port to form the correct TCP server socket if both programs reside on the same operating system. If not, user applications can utilize the VCDI over a local network or the Internet with a valid VCDI server socket.

The VCDI understands two different packet varieties: Passthrough and Packet Forwarding Setup. The structure of a VCDI packet begins with a general header, describing the packet type and size. The type of a packet can be found in its first four bytes, in the form of a 32-bit integer. This field allows the VCDI to distinguish between Passthrough and Packet Forwarding Setup packets. The next four bytes contain another integer describing the number of bytes in the remainder of the packet. The structure of the rest of the packet is dependent upon the value found in the type field.

Byte Index	Type	Name	Description
0	32-bit INT	Type	The type of packet being sent.
4	32-bit INT	Data Size	Number of bytes in the data portion of the packet starting at byte index 8
8	BYTEs	Data	The data that makes up the packet

Table 2. Virtual Cockpit Development Interface packet structure, bytes 0 to 8 (From: Procerus Technologies: "Kestrel Autopilot System," 2006)

a. Passthrough Packets

A Passthrough packet is identified by the number 10 in bytes 0 to 3 (the type field) of a VCDI packet. Passthrough packets are sent from user applications to the Kestrel Autopilot via Virtual Cockpit 2.1, or from the Kestrel Autopilot to user applications via Virtual Cockpit 2.1. Those traveling from a user application to a Kestrel Autopilot contain either commands or a command edit. Passthrough packets heading in this direction contain a 16-bit integer starting at byte 8 that describes the destination address of the intended Kestrel Autopilot. Passthrough packets traveling from a Kestrel Autopilot to a user application will contain either an acknowledgement packet or telemetry data. These packets also have a 16-bit integer starting at byte 8, but in this case it describes the address of the sending Kestrel Autopilot. The next field in a Passthrough packet, located within byte 10, describes the type of the Passthrough packet. There are 250 different packet types, but only five of them are of importance in the context of this thesis: Command packets, Command Edit packets, Acknowledgement (ACK) packets, Negative Acknowledgement (NACK) packets, Telemetry packets and Navigation packets.

Byte Index	Type	Name	Description
8	16-bit INT	Dest/Src Address	The destination if sending or source address if receiving from an autopilot
10	8-bit UCHAR	Kestrel Packet Type	The kestrel packet type
11	BYTEs	Data	The data that makes up Kestrel communications packet

Table 3. Passthrough packet structure, bytes 8 to 11 (From: Procerus Technologies: "Kestrel Autopilot System," 2006)

(1) Command Packets. Byte 10, which describes the Kestrel packet type, contains the number 50 in every Command packet. Command packets are crafted by user applications and sent to the Kestrel Autopilot to order the UAV to carry out some action. The three command types that the VCDI supports are Jump, Loiter and Goto (Procerus Technologies: "Kestrel Autopilot System," 2006). The three have the same packet structure to byte 13. Byte 11 contains an unsigned character (1 byte) that distinguishes between the available commands. Byte 12 contains an unsigned character representing the position of the command relative to the other commands in the UAV's flight plan, and byte 13 describes the total number of commands in the flight plan.

Jump Commands cause the Kestrel Autopilot to move to a different command in the UAV's flight plan. It allows user applications to change the command number that the Kestrel Autopilot is currently executing without modifying the flight plan itself. For example, if a jump to command 1 was issued, the Kestrel Autopilot would move to and execute command 1, then proceed in order with commands 2, 3, etc, regardless of whether or not the commands had already been executed. Jump Command Packets require only one byte of unique input. Byte 14 in a Jump Command Packet describes the command number that the Kestrel Autopilot is instructed to move to and execute.

Byte Index	Type	Name	Description	Value
0	INT	VCDI Packet Type	Indicates a Passthrough packet	10
4	INT	Packet Size	Number of bytes from byte 8 to the end of the packet	6
8	UINT	Destination/Source Address	Contains the destination address of the intended Kestrel Autopilot	1032 (default)
10	UCHAR	Kestrel Packet Type	Specifies the Kestrel Packet type as Command Packet	50
11	UCHAR	Command Type	Indicates a Jump Command	8
12	UCHAR	Command Number	Number describing the position of this command in the flight plan	Varies
13	UCHAR	Total Commands	Total number of commands in the current flight plan	Varies
14	UCHAR	Command Number	Command number to which the Kestrel Autopilot will jump	Varies

Table 4. Complete Jump Command packet structure

A Loiter Command causes the UAV to go to a designated latitude and longitude, and fly in a circle around the specified location. The user can define many of the characteristics of this maneuver including the location of the loiter circle, the altitude of the loiter circle, the flight speed of the UAV to the loiter location, the amount of time the UAV should fly in a circle, and the radius of the loiter circle. The last field in a Loiter Packet is referred to as the “Payload” byte. This byte will be used in the future to indicate what sensors or instruments a particular UAV carries, so it is currently nothing more than a place-holder.

Byte Index	Type	Name	Description	Value
0	INT	VCDI Packet Type	Indicates a Passthrough packet	10
4	INT	Packet Size	Number of bytes from byte 8 to the end of the packet	21
8	UINT	Destination/Source Address	Contains the destination address of the intended Kestrel Autopilot	1032 (default)
10	UCHAR	Kestrel Packet Type	Specifies the Kestrel Packet type as Command Packet	50
11	UCHAR	Command Type	Indicates a Loiter Command	4
12	UCHAR	Command Number	Number describing the position of this command in the flight plan	Varies
13	UCHAR	Total Commands	Total number of commands in the current flight plan	Varies
14	UINT	Altitude	Altitude of loiter	Meters*10
16	UCHAR	Airspeed	Airspeed when flying to the loiter point	(Meters/Second)*2
17	UINT	Loiter Time	Amount of time to loiter	Seconds
19	UINT	Loiter Radius	Radius of loiter circle	Meters

21	FLOAT	Degrees Latitude	Degrees of latitude of loiter circle center	Degrees
25	FLOAT	Degrees Longitude	Degrees of longitude of loiter circle center	Degrees
29	UCHAR	Payload	For future use	N/A

Table 5. Complete Loiter Command Packet structure

A Goto Command causes the UAV to fly to a specified latitude and longitude at a designated speed and altitude. This is the principal command used to get the UAV from one waypoint to the next. Like the Loiter Command, the Goto Command Packet also carries a payload byte as its last field.

Byte Index	**Type**	**Name**	**Description**	**Value**
0	INT	VCDI Packet Type	Indicates a Passthrough packet	10
4	INT	Packet Size	Number of bytes from byte 8 to the end of the packet	17
8	UINT	Destination/Source Address	Contains the destination address of the intended Kestrel Autopilot	1032 (default)
10	UCHAR	Kestrel Packet Type	Specifies the Kestrel Packet type as Command Packet	50
11	UCHAR	Command Type	Indicates a Goto Command	2
12	UCHAR	Command Number	Number describing the position of this	Varies

			command in the flight plan	
13	UCHAR	Total Commands	Total number of commands in the current flight plan	Varies
14	UINT	Altitude	Altitude of waypoint	Meters*10
16	UCHAR	Airspeed	Airspeed when flying to the waypoint	(Meters/Second)*2
17	FLOAT	Degrees Latitude	Degrees of latitude of the waypoint	Degrees
21	FLOAT	Degrees Longitude	Degrees of longitude of the waypoint	Degrees
25	UCHAR	Payload	For future use	N/A

Table 6. Complete Goto Command Packet structure

(2) Command Edit Packets. Command Edit Packets allow user applications to change a command previously uploaded into the flight plan of a Kestrel Autopilot. The structure of these packets is the same as the original issuing command, except that byte 10 contains 53 to identify it as a Command Edit Packet. The data found within the Command Edit Packet simply writes over the indicated command number.

(3) ACK Packets. Acknowledgement Packets are sent from the Kestrel Autopilot to the Virtual Cockpit 2.1 after the successful issuance of an instruction. The Acknowledgement is sent upon the receipt of a legitimate packet; the Kestrel Autopilot does not wait until the instruction contained within the packet is carried out. User applications can have Virtual Cockpit 2.1 forward ACK Packets to them through the VCDI.

(4) NACK Packets. Negative Acknowledgement Packets are sent from the Kestrel Autopilot to Virtual Cockpit 2.1 in the event that the autopilot received an invalid instruction. This generally occurs when packet fields do not align with

specification or when fields contain a reference to a command that does not exist within the autopilot's flight plan. User applications can have Virtual Cockpit 2.1 forward NACK Packets to them through the VCDI.

(5) Telemetry Packets. Telemetry Packets flow from the Kestrel Autopilot to Virtual Cockpit and to any user application that has enabled packet forwarding through the VCDI for Kestrel Packet type 249. These packets contain the UAV's current altitude, velocity, roll, pitch, heading, turn rate, and a host of other metrics describing the UAV's position, vector in three dimensions, and electrical system status. However, Telemetry Packets do not contain GPS data.

(6) Navigation Packets. Like Telemetry Packets, Navigation Packets are full of information describing the status of the UAV and are sent from the Kestrel Autopilot to Virtual Cockpit. Any user application can request Navigation Packets from the VCDI by turning on packet forwarding for Kestrel Packet type 248. Unlike Telemetry Packets, almost every field in a Navigation Packet contains data that was computed using GPS. For example, Navigation Packets include GPS latitude, GPS longitude, GPS altitude, GPS heading, and many other values concerning the position of the UAV and the strength of its GPS signal. User programs requiring latitude and longitude values to function must tap this resource.

b. Packet Forwarding Setup Packets

When packet forwarding is enabled for a specific packet type, the VCDI will send a copy of each packet of that type Virtual Cockpit 2.1 receives from the Kestrel Autopilot to the requesting user program. A Packet Forwarding Setup Packet must first be sent from a user program to Virtual Cockpit 2.1 to start the forwarding service. Bytes 0 to 4 of a Packet Forwarding Setup Packet contain the integer 20. These packets have two unique fields. The Packet ID byte contains a value that identifies the Kestrel Packet type that a request concerns. The following byte contains a 1 if packet forwarding is to be turned on and a 0 if it is to be stopped.

Byte Index	Type	Name	Description	Units
8	8-bit UCHAR	Packet ID	The Kestrel communications packet id number to be configured	N/A
9	8-bit UCHAR	On/Off	Set to 1 to turn on forwarding, 0 turns off forwarding	N/A

Table 7. Packet Forwarding Setup Packet structure, bytes 8-9 (From: Procerus Technologies: "Kestrel Autopilot System," 2006)

Packet ID	Name	Direction	Description
10	Passthrough	Bi	Passthrough packets that can be sent to the autopilot or received from if forwarding is enabled
20	Packet Forwarding Setup	To	Sets up the forwarding tables in the Virtual Cockpit

Table 8. Virtual Cockpit Development Interface packet types (From: Procerus Technologies: "Kestrel Autopilot System," 2006)

4. Ground Station

Users can choose to control their UAV through Virtual Cockpit 2.1, an application utilizing the Virtual Cockpit Development Interface, or manually by remote control (RC). Regardless of the method selected, a wired connection to the UAV is clearly out of the question. Commands issued by the user must be sent from the ground to the Kestrel Autopilot over a wireless connection.

Ground station is the term used to describe the hardware that hosts the user interface to a UAV autopilot. In the case of the Kestrel Autopilot, this consists of either a laptop or desktop computer coupled with a Procerus Commbox, or optionally an RC—Procerus Commbox combination. Procerus Commboxes use the AeroComm AC4490-1000-M3 radio frequency (RF) modem to communicate with the Kestrel Autopilot.

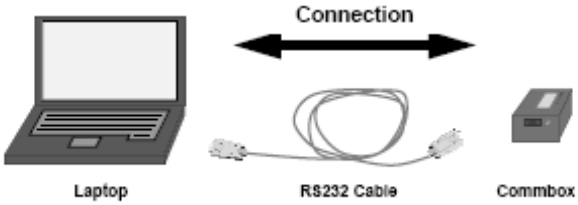


Figure 26. Connecting the ground station computer to a Procerus Commbox to form a Kestrel Autopilot System ground station (From: Procerus Technologies: "User Guide," 2006)

The Commbox has three defined roles within the Kestrel Autopilot System. It is responsible for providing the ground station computer (and relevant user applications) with GPS coordinates detailing the location of the ground station, handling communication between the ground station computer and one or more aircraft" (Procerus Technologies: "User Guide," 2006), and interpreting and sending RC commands to the Kestrel Autopilot.

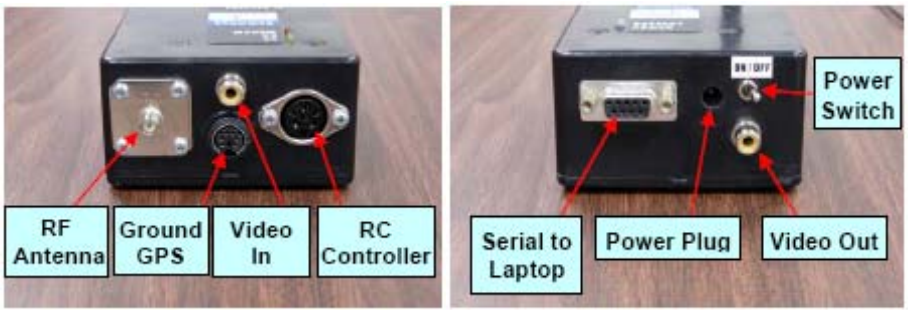


Figure 27. Procerus Commbox ports and connections (From: Procerus Technologies: "User Guide," 2006)

a. Ground Station to Autopilot Communication

(1) Modem Compatibility. The Kestrel Autopilot has a 5-pin modem connection port capable of "supporting serial transistor-transistor logic (TTL) communication" (Procerus Technologies: "Kestrel Autopilot System," 2006). Like the GPS serial I/O port, the Kestrel Autopilot's modem connection port was designed to interface with COTS RF modems. Procerus recommends using the AeroComm AC4490-1000-M3 RF modem because the Kestrel Autopilot was designed to "piggy-back" on it (see Figure 21) through a pin compatible header connection (Procerus Technologies: "Kestrel Autopilot System," 2006). The header connection keeps the Kestrel Autopilot and AeroComm RF modem package together in one compact package. While the RF modem utilized by the Kestrel Autopilot is changeable, the RF modem in the Procerus Commbox is not. These Commboxes come from Procerus with a preinstalled AeroComm RF modem.

(2) AeroComm AC4490-1000-M3 RF modem. According to AeroComm's specification document, the AC4490-1000-M3 RF modem operates at 900MHZ and is "capable of up to 76.8 kbps communications at ranges of over 10 miles line-of-sight" (Procerus Technologies: "Kestrel Autopilot System," 2006). It can operate

in one of two modes: Broadcast or Polling. In broadcast mode, the UAV will send a continuous stream of unprovoked telemetry packets to its ground station. This mode can not be used if there is more than one UAV outfitted with the Kestrel Autopilot operating within 10 miles because the rapid telemetry packet transmissions from two UAVs in broadcast mode would cause too many transmission collisions to function properly. In polling mode, the UAV waits to be queried for a telemetry packet by the ground station before transmitting telemetry data. This mode of operation allows for more than one UAV outfitted with the Kestrel Autopilot to fly at a time, but telemetry values update with higher latency.

b. Ground Station Computer System Requirements

Ground station computers must connect to the Procerus Commbox through an RS-232 serial port. Thus they themselves must have a serial port, or the proper conversion cables and drivers to support connectivity to a serial port. Users issuing commands to the UAV through VC must utilize a ground station computer that also meets the system requirements necessitated by VC.

III. PATH CALCULATION AND PACKET TRANSLATION APPLICATION: OVERVIEW, ARCHITECTURE AND IMPLEMENTATION

A. INTRODUCTION

The Path Calculation and Packet Translation Application version 1 (PCPTAv1) is part of an ongoing project at the Naval Postgraduate School which seeks to implement an autonomous UAV—WSN system designed for military and homeland security surveillance applications. PCPTAv1 is the main contribution of this thesis. In this chapter, PCPTAv1's role within the system will be explained as we take an in-depth look into its organization and functionality.

B. AUTONOMOUS UAV—WIRELESS SENSOR NETWORK SYSTEM OVERVIEW

Confronted with an area of particular interest that is either too vast or dangerous to insert human assets, the UAV—WSN system can deploy to collect environmental sensor data and send it to a base station positioned in a safer locale. Although it is not yet ready to serve in this capacity, the MMALV will eventually contain the hardware to function as a WSN node. There are a number of modes being explored to deliver MMALVs to an area of operations due to its relatively short range, such as attaching several to a larger UAV or dropping them from balloons. Once deployed in the desired layout, this adaptable, meshed WSN begins transmitting sensor data, relayed either by satellite or networking equipment onboard the delivery platform to a ground station miles away. The data streaming back from the WSN is continually evaluated by OTAv1, measuring each sensor reading against environmental thresholds set during deployment. When a contact event is triggered by OTAv1, TRSSv3 activates sensor node cameras at the appropriate times, as determined by the velocity and directional outputs generated by OTAv1. The system administrator can then assess whether or not the contact requires further examination based on the returned sensor readings and uploaded camera pictures. With a few modifications to OTAv1, which will be discussed at length in a later section, contact events will also trigger PCPTAv1 to ask the administrator whether or not to send a UAV to either the location of the activated sensor network or to an estimated

interception point with the contact. This UAV could be the same aircraft functioning as a wireless relay point and perhaps the same vehicle used to deliver the network; or alternatively, serve only as an investigative tool, launching upon contact detection. The sensory capabilities of this UAV would at a minimum entail streaming video, in addition to the various sensors required by the aircraft's Kestrel Autopilot.

In the case where further inspection is requested, PCPTAv1 assesses the UAV's current position and calculates an optimized series of two dimensional waypoints, guiding the UAV to the selected destination. PCPTAv1 packetizes these waypoints according to the VCDI specification and sends them over a TCP/IP connection to the Virtual Cockpit Development Interface where they are routed to the UAV's Kestrel Autopilot via VC. Once onsite, the UAV enters a loiter circle and transmits video of the area below. VC can be used to correct the UAV's position until an adequate visual angle of the contact is acquired.

C. WIRELESS SENSOR NETWORK CONTACT SCENARIOS

As discussed briefly in the prior section, there are three options presented to the PCPTAv1 user upon contact detection. The quality of the sensory data returned by the WSN will determine the accuracy of the initial contact classification. If the network administrator is satisfied with the data upon which the classification was based as well as the classification itself, a UAV need not be sent to investigate the contact further. Thus the option to take no further action is included in the PCPTAv1 GUI.

1. Contact Interception

Some WSN contacts will not emit distinctive sensory readings making classification difficult. Others may not traverse through the network on an ideal heading to capture useful TRSSv3 images. Still others may be of particular interest and necessitate a tag to monitor the contacts movements and actions. In each of these cases, a UAV must be used to capture video of the contact. The contact, however, will not remain motionless to wait for the UAV's arrival.

With the output returned by a slightly modified OTAv1, PCPTAv1 can extrapolate the WSN contact's path and compute an optimal flight plan to guide a designated UAV on an intercept course. This option is presented to the user in the

PCPTAv1 GUI upon contact detection. Before sending a UAV, PCPTAv1 first evaluates whether the contact can be caught given the physical limitations of the UAV in question.

2. Operational Support

When many contacts enter the WSN, each exhibiting a different trajectory, it becomes inefficient to associate a separate UAV with each contact. Rather than follow a particular contact, situations such as this call for one or more UAVs to deploy in the vicinity of the sensor network to provide visual support for a potential engagement. PCPTAv1 incorporates the functionality to contend with this scenario, by offering the user the option to send a UAV to the center of the instigating WSN.

D. APPLICATION DEVELOPMENT

1. Programming Language

PCPTAv1 was written in C++ for two reasons. The author's familiarity with this language was certainly a strong motivation. Also, Procerus Technologies wrote sample C++ code to illustrate how to use the VCDI. With permission from Procerus, the author began developing PCPTAv1 from the framework supplied by this code.

2. Development Software

Microsoft Visual Studio 2005 Professional Edition was used to develop PCPTAv1. It is recommended that any future compilations of this code be carried out using Microsoft Visual Studio as well, due to this code's susceptibility to misinterpretation and reliance upon Microsoft Foundation Classes. The number of bytes used to represent some of the variable types found in this code is not standard across different compilers. The packets sent from PCPTAv1 depend on the standards within Microsoft Visual Studio to associate the expected number of bytes with each variable. Without compliance to this standard, any packets sent to the VCDI will be discarded and never reach the UAV.

E. DESIGN CONSIDERATIONS AND ASSUMPTIONS

1. Software/Hardware Dependencies

a. Hardware Requirements

The hardware requirements outlined in both the Virtual Cockpit 2.1 and Ground Station sections of Chapter II also apply to any system running PCPTAv1. No UAV utilizing the Kestrel Autopilot System can fly autonomously without either VC or a

Procerus Commbox. PCPTAv1 relies on VC to host the VCDI and relay its packets to the Kestrel Autopilot. Without a Commbox, communications between user applications (including VC) and the UAV would not be possible.

PCPTAv1 is intended to be only the first iteration in a series of such applications, and so developers will be interested in the hardware requirements necessary to compile the code. Using Microsoft Visual Studio 2005 Professional Edition (as recommended) requires a system with at least the following (Microsoft Corporation, 2006):

- 1 GHz processor
- 256 MB of RAM
- 2 GB of available hard drive space
- Microsoft Windows 2000 Professional Edition (or later)

b. Software Dependencies

For any system to run PCPTAv1 in its current form, the following software is required:

- Microsoft Windows XP
- Microsoft Visual Studio 2005 Professional Edition
- Virtual Cockpit 2.1

2. Assumptions

a. Autopilot Telemetry Accuracy

PCPTAv1 depends on the accuracy of the telemetry and navigational packets sent from the Kestrel Autopilot to the ground station in all of its calculations. This is the application's sole source of data as to the active UAV's vector and orientation. The Kestrel Autopilot System has proven to be extremely reliable in this regard, and so it is assumed that such behavior will continue.

b. Flight Plan Considerations

The flight paths that PCPTAv1 creates are close to optimal in two dimensions. Once uploaded to the Kestrel Autopilot, getting the UAV to the waypoints contained in each flight plan is not PCPTAv1's responsibility. This creates several

problems. First, if the GPS used by Kestrel Autopilots for navigation is not differential, which happens to be quite expensive at this time, its GPS readings will only be accurate to a few meters. While accuracy of this level is not necessarily essential in this application, it is important to note.

Wind resistance is not considered in PCPTAv1 calculations. Intercept flight plans depend on the precise execution of waypoints based on the maximum velocity of the UAV, which is inputted by the user. Wind can have a large effect upon the path and velocity of micro-UAVs, and could grossly affect a UAV's ability to reach an intercept location at the intended time. In high winds, many micro-UAVs exhibit erratic behavior and are not capable of flying in a straight line, making the optimization efforts of PCPTAv1 worthless.

The flight plan generation algorithms in PCPTAv1 do not account for physical obstructions. Kestrel Autopilot is outfitted with a barometer, and gauges its altitude by the difference between its current the pressure and a ground pressure reading. While it can adjust to slight inclines, a sharp, protruding mountain could spell disaster for the Kestrel Autopilot. PCPTAv1 will happily slam the UAV into the side of a mountain or building that falls along the computed path to its destination.

3. Operational Considerations

The UAV—WSN system is being designed with deployment in mind. Its COTS hardware components are easily obtainable and inexpensive. The training required for administrators will be minimal since most of its operational challenges are abstracted from the user and handled autonomously. Where the user must interact with the system, the user interface has been made as simple as possible. Kestrel Autopilot controlled UAV platforms can be upgraded, modified and swapped while retaining the support provided by PCPTAv1 and VC. The scalability of this solution is as easy as adding or removing UAVs and WSN clusters to an operational area. With MMALV nodes, the mobility and meshed communications of this system's WSN make it extremely adaptable to the characteristically unpredictable nature of most operational environments.

F. UAV—WSN SYSTEM COMPONENT APPLICATION INTEGRATION

While TRSSv3 and OTAv1 were developed simultaneously with collaboration in mind, the same cannot be said for PCPTAv1. PCPTA was produced the year following these applications. Due to the author's inability to weigh in on OTAv1 design objectives, some modifications must be made to this program before it can provide PCPTAv1 with the input it requires.

OTAv1 is limited to one of three WSN deployment scenarios. Each of these scenarios require sensor nodes be placed along a road or corridor. In the absence of such a formation, OTAv1 will recognize a contact's presence, but will not be able to measure its speed or assess its direction of travel. Any system suffering these restrictions would not be viable for real-world use. OTAv1 must be improved upon to include support for a nebulous, adaptable WSN layout.

OTAv1 was designed specifically to interpret a Crossbow MBR410 Base Station serial data stream from a MSP410 Mote Security System. It is unknown at this time what WSN hardware the UAV—WSN system will use; however, OTAv1's use of the MSP410 Mote Security System has had a significant impact upon the node data it expects and utilizes in its calculations. OTAv1 had only IR sensors and magnetometer readings at its disposal. Without mote GPS support (as is the case in the MSP410 package), OTAv1 is forced to assess contact direction of travel relative to the position of its sensor motes. In the straight road scenario, for example, OTAv1 outputs whether a contact is moving left or right down the road. It does not provide users with the contact's bearing, which is an integral piece to any interception calculation. The final version of OTA must be able to retrieve GPS data from MMALV WSN nodes, and use it to both track the location of WSN clusters and calculate contact bearing.

TRSSv3 is not expected to provide any input directly to PCPTAv1. It activates a camera to take pictures of a WSN contact at the right times. These pictures are viewed by the user to assist in contact classification, and the quality of this data will weigh into the user's UAV deployment scenario selection. However, this application is certainly not in its final form, as most of the hardware will have to change to port TRSSv3 functionality to the MMALV. The MMALV is outfitted with a CMC-08P micro color

CMOS camera, and certainly could not carry the weight of the Creative WebCam used in the current TRSSv3 implementation. Depending upon the size and weight of commercially available satellite transceivers, the pictures taken may have to be sent to the base station before they are uploaded to a web server, as opposed to the current implementation scheme where they are sent through a satellite connection to the Internet.

G. PCPTAV1 GRAPHICAL USER INTERFACE

The PCPTAv1 GUI is what the user sees while using this application. It is the interface through which the user controls the program. Figure 28 depicts the PCPTAv1 GUI, and should be used as a reference through the following description.

There are several distinct sections, each enclosed in a grouping box with a blue title. The “Zero Pressure” section, along with its underlying code was created by Procerus Technologies. The “Packet Forward Selection,” “Acknowledgements” and “Telem and Nav Info” sections were originally produced by Procerus Technologies, but were modified by the author, and The UAV-WSN System section is solely the work of the author.

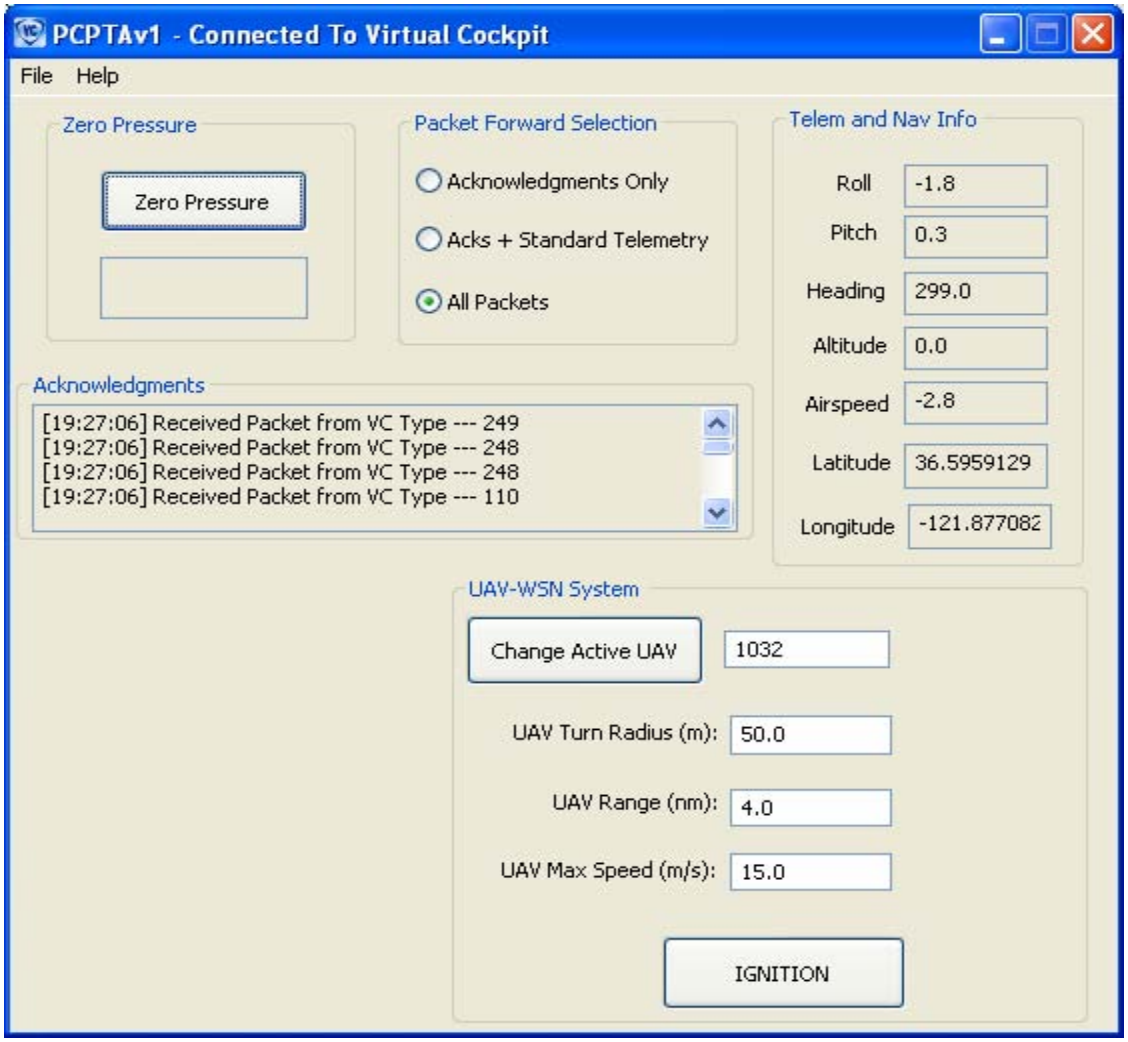


Figure 28. PCPTAv1 GUI

1. Title Bar

One of the first actions taken by PCPTAv1 upon execution is to create a TCP/IP connection with the VCDI. The most common causes of connection failure are due either to the user not starting an instance of VC, or use of the wrong VCDI destination socket within PCPTAv1. Regardless of the cause, the user must be made aware of such a failure since PCPTAv1 cannot send or receive packets from a Kestrel Autopilot without a connection to the VCDI. The title bar is utilized to relay the status of this connection to the user.

2. Zero Pressure Grouping

Before takeoff, the user must zero the Kestrel Autopilot's pressure sensor to associate the correct barometer reading with ground level. This can be done in VC by

depressing the Zero Pressure button in the preflight tools module (see Figure 25). This button was included in PCPTAv1 to show future PCPTAv1 developers how to lift VC functionality off of its user interface. PCPTAv1 can communicate with VCDI over a TCP/IP connection, so there is no guarantee (thanks to modern networking technology and the Internet) that users will have physical access to the VC instance hosting VCDI. This button is the beginnings of the conversion process that will eventually instill within PCPTA all the options the user requires to control the takeoff, flight and landing of a UAV.

3. Packet Forward Selection Grouping

On Broadcast mode (default) Kestrel Autopilot will flood VC with a rapid stream of acknowledgement, telemetry and navigational packets. The Packet Forward Selection grouping offers the user three alternatives for controlling the flow of these packets from VC to PCPTAv1. Selecting any one of these options causes a Packet Forwarding Setup packet (see Table 7) to be sent to VC. The "Acknowledgements Only" radio button will cause VC to allow only packet receipt acknowledgements to flow from the active Kestrel Autopilot to PCPTAv1, while the "Acks + Standard Telemetry" radio button allows only packet receipt acknowledgements and telemetry packets. The "All Packets" radio button lets all packets flow to PCPTA and is the default setting.

The user's selection in this grouping can affect the ability for other parts of the application to function. The "All Packets" radio button must be selected for PCPTAv1 to receive the information it requires to address a WSN contact. While either of the other two radio buttons is selected, PCPTAv1 does not receive Navigation packets from the Kestrel Autopilot. Meaning the application will not possess the GPS coordinates describing the UAV's current location.

4. Standard Telemetry Information Grouping

This section displays the active UAV's orientation, airspeed, and location in real-time. The roll, pitch, heading, altitude, and airspeed are taken from Kestrel Autopilot Telemetry packets. For these values to refresh, either "Acks + Standard Telemetry" or "All Packets" must be selected in the Packet Forward Selection grouping. As stated above, the latitude and longitude values are taken from Kestrel Autopilot Navigation

packets. For the longitude and latitude indicator to display an updated UAV position, "All Packets" must be selected in the Packet Forward Selection grouping.

5. Acknowledgements Grouping

The Acknowledgement window displays a rough description and the time of receipt for any packet received by PCPTAv1. The user should expect to see command packet acknowledgements from the active Kestrel Autopilot, Telemetry packet indicators, and Navigation packet indicators appear here, depending upon the selected radio button in the Packet Forward Selection grouping.

6. UAV-WSN System Grouping

This grouping contains the user inputs necessary for a UAV to participate in WSN operations. and a start button that causes PCPTAv1 to search for WSN contact reports from OTAv1. The "Change Active UAV" button takes the user supplied input in its corresponding edit box and uses this number as the new Kestrel Autopilot address in all of PCPTAv1's operations, including the data displayed in the "Telem and Nav Info" grouping. This allows the user to direct more than one UAV in the operational area if necessary.

When the "IGNITION" button is pressed, PCPTAv1 searches for WSN contact reports from OTAv1. If a contact report is found, PCPTAv1 will ask the user whether to send the active UAV after the contact, send the active UAV to the instigated WSN cluster, or take no action (see Figure 29). Picking either the first or second option will call into play the other inputs in the "UAV-WSN System" grouping. The "UAV Turn Radius," "UAV Range" and UAV Max Speed" must be set before the "IGNITION" button is pressed. They each describe a different UAV attribute, and can be used to scale the flight plans produced by PCPTAv1 to any platform utilizing the Kestrel Autopilot System.

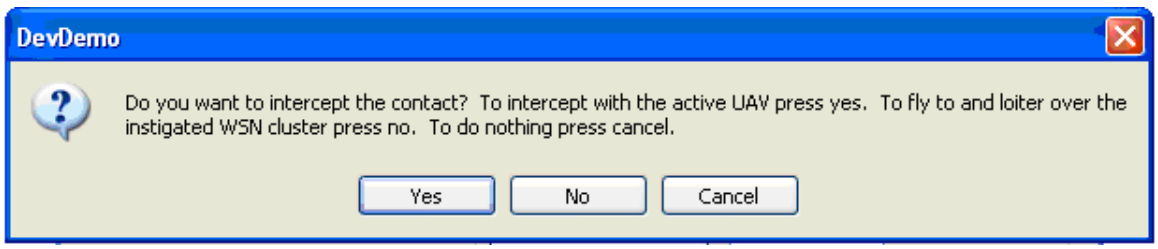


Figure 29. User options upon PCPTAv1 detection of a WSN contact report

H. PATH CALCULATION AND FLIGHT PLAN PRODUCTION

The two primary goals of this thesis are to be able to construct two dimensional optimized UAV flight plans based on WSN output, and to build the functionality required send a flight plan to the Kestrel Autopilot via the VCDI. This section describes the specifics of the latter. Flight plans are calculated in PCPTAv1 by the PathPlanner class (see Appendix C).

1. Calculation Inputs

a. User GUI Inputs

Since the user is free to install a Kestrel Autopilot within any flyable UAV chassis, its physical attributes must be modifiable. The user is responsible for these inputs within the PCPTAv1 GUI. These inputs were introduced in the previous section during discussion of the "UAV-WSN System Grouping." The UAV's turn radius is required to tune flight plan turns to the active UAV. A larger turn radius will produce flight plans with wider turns. Users must be careful not to underestimate this input, as it will result in a highly inefficient flight plan. The UAV's range is used to ensure that a destination is not set outside of ground station transmission range or further than the battery life of the vehicle will allow. The UAV is sent at its maximum speed to investigate WSN contacts, regardless of the scenario chosen. If sent to the WSN, the UAV must be onsite as quickly as possible to ascertain the situation so an appropriate response to the contact can be mounted. In interception scenarios the contact's path is estimated, and the further this estimation is extrapolated, the less accurate this point will be. Therefore, the interception point reachable by the UAV, if interception is possible, at its top speed will be the most accurate attainable.

b. Kestrel Autopilot Telemetry and Navigational Inputs

On its default setting, the "All Packets" radio button is selected on the PCPTAv1 GUI. This provides the application with a host of data detailing the position, orientation and velocity of the active UAV. Of this data, the latitude and longitude is used in flight plan calculations as the starting point for any computed path. The current heading of the UAV is also taken into consideration, as a turn may or may not be necessary to put the UAV on the correct heading to its destination.

c. WSN/OTAv1 Inputs

From the WSN, or peripheral applications parsing the WSN output such as OTAv1, PCPTAv1 requires several pieces of information to formulate an appropriate UAV flight plan. When a contact is detected, PCPTAv1 must receive the latitude and longitude of the triggered WSN cluster. This data is used as either the starting point of the contact's estimated path in intercept calculations or as the UAV's final destination in operational support scenarios. The bearing and velocity of the contact will only be utilized if the user chooses to send a UAV after the contact; however, PCPTAv1 poses the user with this decision after WSN output is received. Therefore, this information must always be included. In its current state, PCPTAv1 expects to find contact data in a text file named "contact." The existence of this file in PCPTAv1's local directory is used to indicate a detection event. Figure 30 displays the format in which this data should appear, where the first number is the contact's heading (in degrees), the second is contact velocity (in meters/second), and the third and fourth numbers are the (degrees) latitude and (degrees) longitude respectively. The text file is necessary because these inputs cannot at this time be gleaned from the output of OTAv1 or any commercially available WSN system.

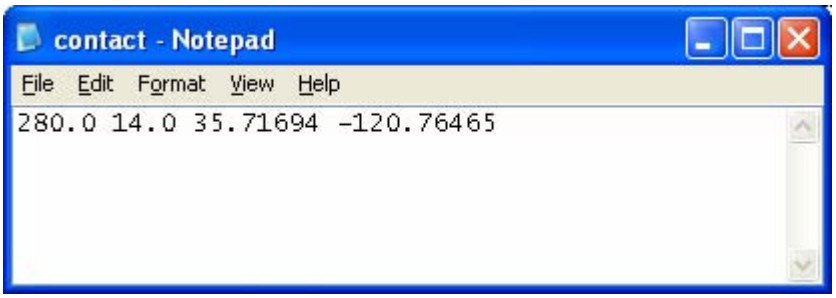


Figure 30. WSN output in "contact.txt"

2. Sensor Network Investigation Scenario Flight Plan Calculation

This scenario is realized when PCPTAv1 detects a WSN contact report from OTAv1 and the user selects to send the active UAV to the WSN cluster that made the detection. The final waypoint is set at the latitude and longitude of the instigated WSN cluster. The first PathPlanner operation carried out is to measure the distance between the UAV's current location and its destination using the haversine formula. This equation was published by R. W. Sinnott in Sky and Telescope (1984). Since this

formula is based on a spherical model, it does not account for the elliptical shape of the Earth, and may not be the most accurate formula to use in some cases. It is, however, quite accurate (all navigational models have some error) over small distances, making it ideal for use with micro UAVs. The haversine formula and its C++ implementation:

$$\operatorname{haversin}\left(\frac{d}{R}\right) = \operatorname{haversin}(\Delta\phi) + \cos(\phi_1)\cos(\phi_2)\ \operatorname{haversin}(\Delta\lambda)$$

Figure 31. The haversine formula, where $ø_1$ and $ø_2$ are latitudes, λ_1 and λ_2 are longitudes, *d* is the distance between points 1 and 2, and *R* is the radius of the sphere upon which the points reside (From: Wikipedia: "Haversine Formula," 2006)

$$d = R\operatorname{haversin}^{-1}(h) = 2R\arcsin\left(\sqrt{h}\right)$$

Figure 32. The haversine formula solved for *d*, where *h* denotes "haversin(*d*/*R*)," (From: Wikipedia: "Haversine Formula," 2006)

The measureDistance function in the PathPlanner class solves the rightmost side of the equation shown in Figure 32: "dist=2*asin(sqrt(pow((sin((lat1-lat2)/2)),2.0) + cos(lat1)*cos(lat2)*(pow((sin((lon1-lon2)/2)),2.0))));" where input coordinates and output distance are in radians, and North and West coordinates are treated as positive.

Before any further computations are completed, the range of the active UAV is compared to the distance between it and the instigated WSN cluster. If the destination is reachable, its latitude, longitude, and distance from the active UAV's current position are used to determine its bearing from the UAV.

The initial bearing to the WSN cluster must be calculated to determine whether or not the UAV must turn, and if so, how hard. Figure 33 depicts a pseudo code representation of this calculation, and is followed by the actual code used in PCPTAv1 (Figure 34).

```
dlat = lat2 - lat1
dlon = lon2 - lon1
y = sin(lon2-lon1)*cos(lat2)
x = cos(lat1)*sin(lat2)-sin(lat1)*cos(lat2)*cos(lon2-lon1)
if y > 0 then
if x > 0 then tc1 = arctan(y/x)
if x < 0 then tc1 = 180 - arctan(-y/x)
if x = 0 then tc1 = 90
if y < 0 then
if x > 0 then tc1 = -arctan(-y/x)
if x < 0 then tc1 = arctan(y/x)-180
if x = 0 then tc1 = 270
if y = 0 then
if x > 0 then tc1 = 0
if x < 0 then tc1 = 180
if x = 0 then [the 2 points are the same]

if x = 0 then tc1 = 270

if y = 0 then

if x > 0 then tc1 = 0

if x < 0 then tc1 = 180

if x = 0 then [the 2 points are the same]
```

Figure 33. Pseudo-code representation of the bearing calculation from one known point to another (From: The Math Forum, 2001)

```
newBearing=atan2(sin(lon2-lon1)*cos(lat2),
cos(lat1)*sin(lat2)-sin(lat1)*cos(lat2)*cos(lon2-lon1));
```

Figure 34. C++ code to find the bearing from one known point to another (After: Movable Type Scripts, 2006)

3. Intercept Scenario Flight Plan Calculation

a. Setup

As in the previous scenario, the distance between the active UAV and the WSN cluster reporting the contact is computed using the haversine function. This will be used later in the computation. Two arrays are designated to hold estimated contact position coordinates; one is used to hold latitudes and the other longitudes. The size of these arrays is determined by the equation: "(V*3600*2)/5," where V is the contact velocity (m/s). Memory is allotted to track the contact's estimated path for two hours, in five meter increments. Accuracy greater than 5 meters along an intercept path is irrelevant thanks to the vantage point of the UAV and the resultant ground coverage in its video image.

Since this model expects the contact to maintain a constant speed and direction from the time of detection, it will become highly inaccurate with the passage of time. If the active UAV cannot intercept a contact within two hours, the application will not send it. The chances of the contact remaining along the same path at the same speed for over two hours are extremely small in the absence of a long, straight road.

b. Feasibility Check

Before jumping into an exhaustive search, a few checks are conducted to assess the feasibility of finding an appropriate interception point. If the contact is moving away from the active UAV in x and y (with reference to Figure 35), and moving faster than the UAV's maximum speed, the calculations will never return a reachable result based on these inputs. The case where the contact is moving away in x and y (with reference to Figure 35) and is at the time of detection located outside the range of the UAV will also set the unreachable flag. These checks do not account for every potential situation where the interception point is unreachable or nonexistent. All other cases are handled during the estimated path polling operation.

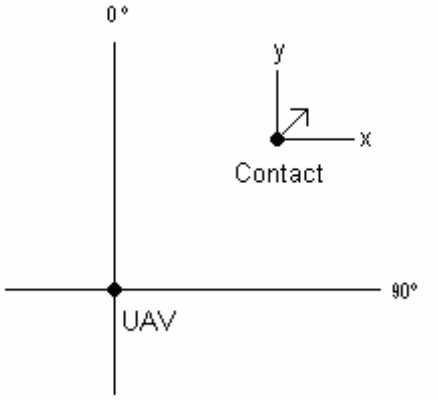


Figure 35. Contact bearings relative to the UAV in the X and Y directions

c. Estimated Path Polling

The polling operation checks along the estimated contact path (every 5 meters) for the closest interception point to the active UAV's current position. This is computed by tracking the time it would take the contact to reach each point along its path, and comparing it to the time it would take the active UAV to reach the same location. The first point found, starting from the current location of the contact, that

would take the active UAV less time to get to than the contact becomes the initial destination. Each point along the contact's path is generated and checked, to avoid generating unexploited data.

Path polling is not the most efficient way to solve the interception problem. However, this implementation computes accurate results with an overhead that is easily handled by any modern system. The inefficiency lies in the brute force method it uses to attain its result. The most efficient solution would involve adding a nonlinear equation solver to the application libraries, and solving the system of nonlinear equations detailed in Chapter IV. These equations are used to prove the accuracy of PathPlanner output. PCPTAv1 does not currently include the requisite libraries to solve them, as program efficiency is outside the scope of this thesis.

d. Initial Bearing and Destination to Contact Interception

Using the center of the instigated WSN cluster as the starting location along the contact's projected path, its bearing and a distance interval of 5 meters are used to generate new latitudes and longitudes in the manner detailed by Figure 36. The time it would take the contact to reach the generated point along its path is assessed by tracking the time through each five meter increment. Given the contact's velocity V (m/s), this equation is "Seconds=5.0/V." The distance is then measured between the generated point and the active UAV's current location using the haversine formula. The time it would take the UAV to travel this distance is found by this equation: "distance/(UAV maximum speed)," where distance is in meters and the UAV's maximum speed is in meters/second. As mentioned previously, the time it would take the contact to reach the generated point along its path and the time required by the UAV to reach this destination are compared. If the UAV can get to the generated location before the contact, its initial destination and bearing have been found.

$$lat2=\sin^{-1}(\sin(lat1)*\cos(d)+\cos(lat1)*\sin(d)*\cos(tc))$$
$$lon2=((lon1-\sin(\sin(tc)*\sin(d)/\cos(lat2))+pi)\%(2*pi))-pi$$

Figure 36. This equation finds the point (lat2, lon2) that is a distance d (nm) from point (lat1, lon1) on the true course tc (radians), where coordinates are in radians and the "%" operator denotes modulus (After: Williams, 2004)

e. Final Bearing and Destination to Contact Interception

The time it will take the active UAV to turn toward the initial contact interception point have yet to be factored into the calculation. To remedy this situation, PCPTAv1 runs the initial destination through the same functions it uses to optimize path construction. A flag is set to stop these functions from writing waypoints based on the initial destination. The pathDecide function is run with an input of 1, and returns a 1 if a turn is required and a 0 if not. If a turn is required, the makeTurn function is called with an input of 1 to determine how much of a turn is required. These functions are discussed at length in the following section. Turns are assumed to be a perfect circle, and so the additional time required can be evaluated using the UAV's maximum speed (m/s), the UAV's turn radius (m), and a fraction describing the extent of the turn, where 1 translates to a full 360 degree turn: "Seconds=((2*pi*turnRadius*2) /maxSpeed)*(Turn Fraction)." Based on the velocity of the contact, it can be determined how much further along its estimated path the interception point must be moved due to the turn. The turn time is reinterpreted as a distance with the equation: "contact distance traveled=UAV turn time*contact speed," with inputs of seconds and meters per second respectively. The final interception point is then generated according to Figure 36. This calculation concludes by checking to ensure the contact is still possible to intercept after factoring in the turn.

4. Optimized Path Construction

Optimized UAV paths in two dimensions consist of circular turns with minimized radii and straight lines (see Figure 37). This is the guiding principle upon which PCPTAv1 bases flight plan production. Destination calculations are scenario dependent; however, scenarios rely on the same logic to guide the active UAV along a near optimal path from its current location and bearing to its final destination. The pathDecide function oversees general flight path construction, the makeGotoBeforeTurn and makeTurn functions handle turns where applicable, and the makeGotoFlyStraight function guides flight to the final destination. This function is the last called in every flight plan, sending the UAV into a circular loiter over its final destination to await further commands from the user. These functions are all members of the PathPlanner class.

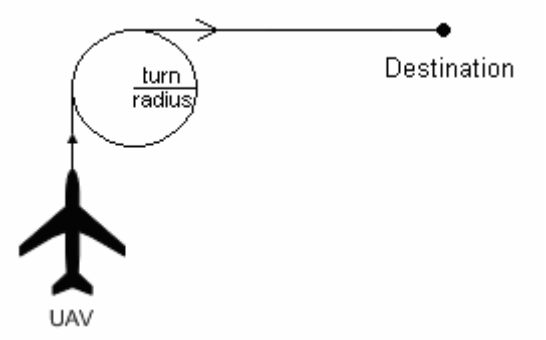


Figure 37. Optimized flight path in two dimensions

a. General Path Construction: pathDecide Function

With reference to Figure 38, there are four zones relative to the current heading of the active UAV. Every destination falls within one. The pathDecide function determines to which zone a given destination belongs and takes the appropriate action. Destinations requiring less than a five degree turn (the red area shown in Figure 38) are fed directly to the makeGotoFlyStraight function, which creates a flight plan consisting solely of the destination waypoint. PathPlanner guidance is not necessary for destinations directly in front of the UAV. Destinations falling outside a distance twice the turn radius of the active UAV, that are within 45 degrees on either side of the UAV's current bearing (green area in Figure 38) are dealt with in the same manner. The Kestrel Autopilot is quite capable of reaching any point requiring less than a 45 degree turn in an efficient manner, given plenty of time and space to turn.

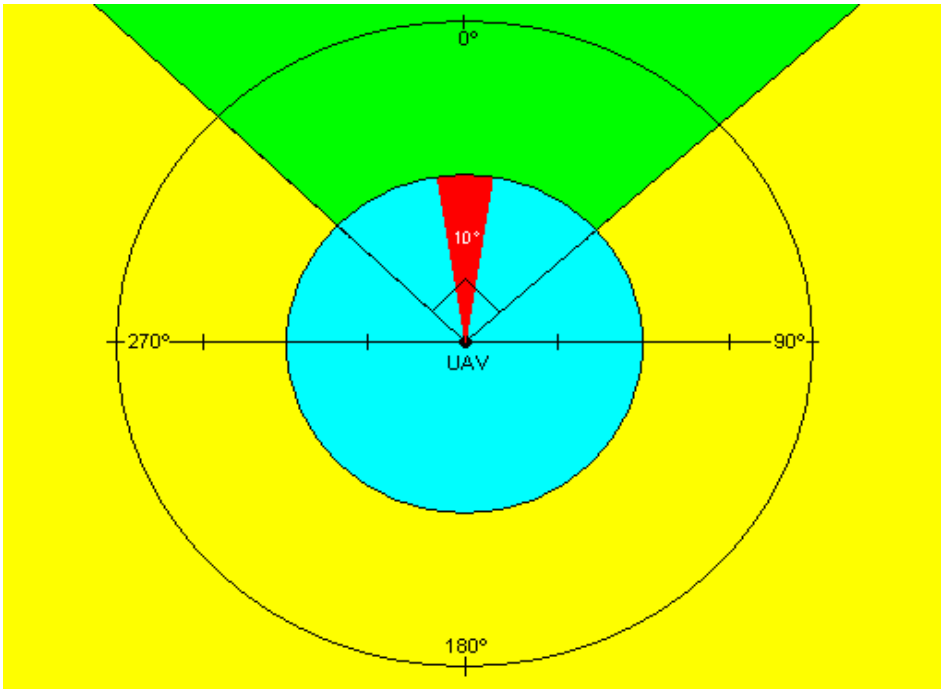


Figure 38. Destination zones relative to the active UAV (bearing 0 degrees). The outer circle has a radius 4 times the turn radius of the UAV.

Figure 39 shows how some destinations falling within a distance twice the turning radius of the UAV from its current position are unreachable. For this reason, destinations falling within the blue area in Figure 38 are handled first by the makeGotoBeforeTurn function followed by the makeTurn function. The latter creates a waypoint directly in front of the active UAV at a distance of four times its turn radius, ensuring that the subsequent turn will be able to reach any destination within the blue area surrounding the UAV's original position (in Figure 38).

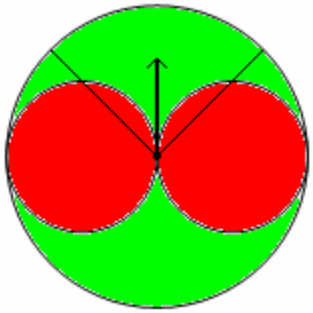


Figure 39. Turning from its current position, the green area represents UAV reachable space. The outer circle has a diameter equal to 4 times the UAV's turn radius.

The remaining zone (yellow area in Figure 38) includes all destinations outside a distance twice the active UAV's turn radius and requiring greater than a 45 degree turn. These are handled with a call directly to the makeTurn function since any point within this zone is reachable with a turn from the UAV's current position (see Figure 39).

b. Turn Optimization: makeTurn Function

When a turn is deemed necessary within PCPTAv1, the makeTurn function is called. This function first decides upon the direction of the turn. Any destination bearing between 180 and 315 degrees relative to the UAV's current heading results in a left turn, while destinations bearing from 46 to 180 degrees (inclusive) result in a right. The forward-looking 90 degree sector is left out of this decision because makeTurn is not called for turns of less than 45 degrees. Eight points are then generated to represent the turning circle. Traveling around the circle, the UAV must make 45 degree turns to get to each successive point. A check is performed at each point to determine the most efficient "break-off" location. The break-off point is where the UAV stops turning and proceeds straight to the destination. The criterion for selecting the break-off location is based upon whether turning toward the next point around the circle would bring the final destination to bear. In the case of a right turn (see Figure 42), the break-off point is found if the destination falls within a range of 0 to 45 degrees relative to the UAV's heading around the circle. The opposite is true for left turns, where this range becomes 0 to 315 degrees relative.

Turning circles must be represented by a series of points due to the limitations of the Kestrel Autopilot System. The Kestrel Autopilot does not support commands entailing a specific turn rate. In other words, one cannot craft a packet instructing the UAV to fly at a given bearing drift over a specific distance. The Loiter Command is the only option offered within the Kestrel Autopilot System for flying the UAV in an autonomously controlled circle. The loiter is controlled by a time input, which could translate to various distances around the circle depending upon the effects of wind. Therefore this command cannot be relied upon to execute optimal UAV turns.

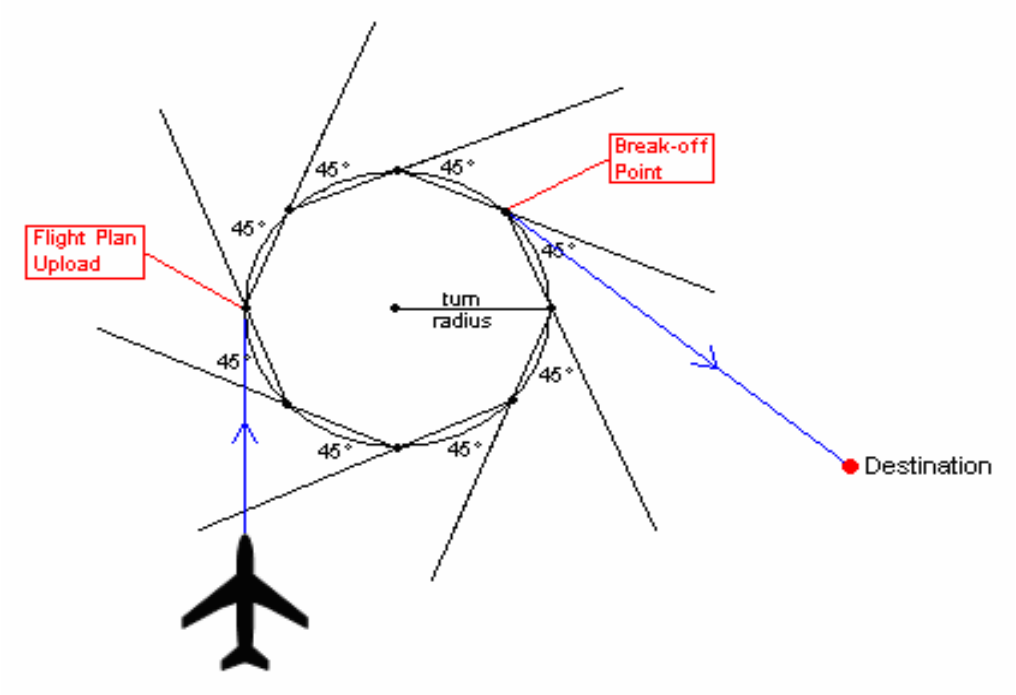


Figure 40. PCPTAv1 turning implementation

The turning calculation can be made more accurate by constructing a circle with more than 8 points. Greater granularity within the model would enable a break-off point to be chosen that is closer to the optimal location: where the tangent line from the turn circle intersects the destination point (see Figure 41).

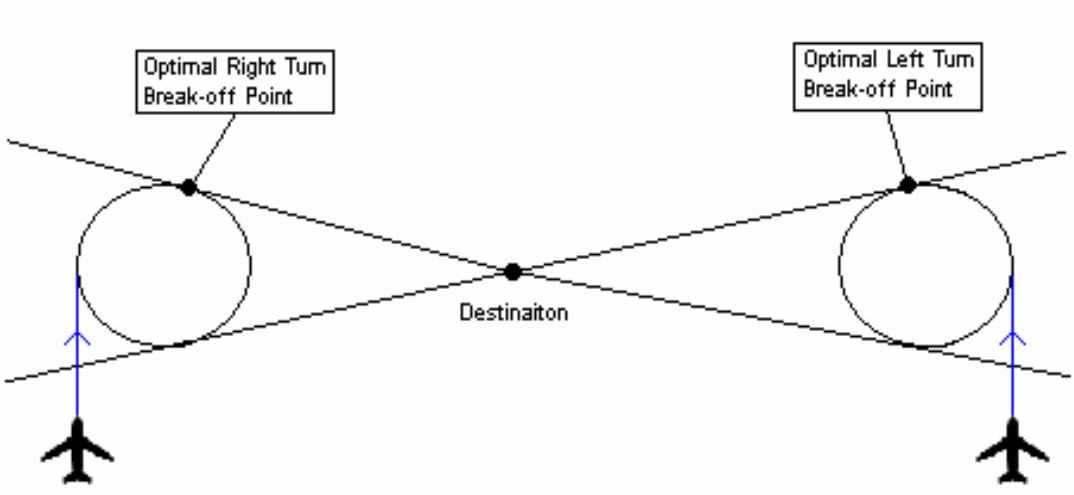


Figure 41. Optimal turns are ended at the first point where the tangent line of the turn circle intersects the destination.

5. Output

The PathPlanner class writes waypoints to a text file called "dat.txt." The flight plan can then be utilized by the DevDemoDlg class to construct and send command packets to the Kestrel Autopilot. The flight plan is passed through a text file to promote modularity within PCPTAv1's development; specifically between the flight plan calculation and packetization processes. This makes it easier to troubleshoot compilation or logical issues and the code simpler to read. Additionally, the constructed flight plan can be viewed by the user without the aid of VC.

Figure 42 displays a typical flight plan produced by the PathPlanner class. Each line contains the data to construct one packetized command, and the type of the command is dictated by the first number. The following numbers are command specific values, and are covered at length in the Background (Chapter II) and Appendix A.

Developers must be careful when manually modifying the "dat.txt" file to adhere to the Kestrel Autopilot System specification. Many values must be multiplied by a specific constant before submitted to the Kestrel Autopilot. For example, the Kestrel Autopilot System manual lists the units of the altitude value in each command packet as "Meters * 10." To send the UAV to an altitude of 100 meters, an unsigned integer with a value of 1000 must be placed in the packet.

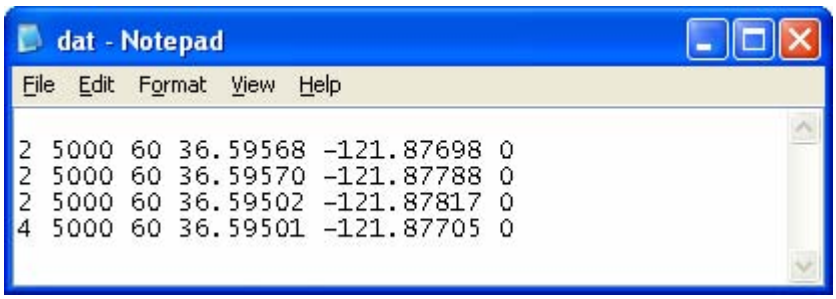


Figure 42. PCPTAv1 flight plan output. The first three lines are Goto commands, while the fourth is a loiter command (as indicated by the first number on each line).

I. VIRTUAL COCKPIT DEVELOPMENT INTERFACE PACKET CRAFTING

Once a flight plan has been created, it must be packetized according to the VCDI specification. This specification is covered at length in both the Background chapter and

Appendix A. The flight plan at this point resides in a text file (see Figure 42) named "dat," and each line within this text file contains the data for one packet. The values in each line must be properly cast and sent to the VCDI socket in a specific order.

Packets are constructed within PCPTAv1 using unique "struct" data structures to hold each packet type. The lines within "dat.txt" are read one at a time in the OnBnClickedPumpData function. As they are read, each line is passed to the sendPacket function to be parsed, formatted and sent to the VCDI.

1. GoTo Command Packet Crafting Example

To ensure clarity, we will step through the packetization of a GoTo command. Table 6 provides a byte by byte breakdown of a GoTo command packet. The GoToPacket struct, which can be viewed in Appendix C, is used to store each value in the order they are to appear in the packet so that its contents can be copied in one operation. This order is determined by the variable listing sequence in the struct definition. Struct definitions can be found in DevDemoDlg.h (see Appendix C). The first value filled within the GoToPacket struct is the packet destination, which is taken from GUI input (UAV-WSN System grouping). Next, the packet type is always 50 for GoTo packets since they are commands. The command type is also hard-coded because it identifies the packet as a GoTo command. The command type will always be 2 in GoTo packets. The next two struct values contain the current command number and total number of commands in a flight plan. Each time the sendPacket function is called from OnBnClickedPumpData, the variable keeping track of the current command number is incremented. Before sendPacket is entered, countCommands is called in OnBnClickedPumpData. This function counts the number of total packets, and hence, the number of commands that are in the flight plan. The remaining struct variables are command specific. For GoTo packets, these are altitude, airspeed, latitude, longitude and payload. These values are taken directly from the flight plan text file. The GoToPacket struct is then copied into an unsigned character array within an sVCPacket struct. The sVCPacket contains two variables of its own that precede the aforementioned array. Think of the GoToPacket struct as the packet payload and the sVCPacket as its header. The first is the VC packet type, which is always 10 for Passthrough packets. The next

variable, data size, contains the number of bytes copied from the GoToPacket struct. The sVCPacket is then sent to the VCDI socket by the SendData function.

2. Viewing PCPTAv1 Flight Plans on a Geo-Referenced Map

Once received by the active Kestrel Autopilot, the user can view uploaded waypoints using the geo-referenced map pane in the VC GUI, as seen in Figure 43. Before the map can be populated, the "download" button must be pressed to pass the current flight plan from the Kestrel Autopilot to VC.

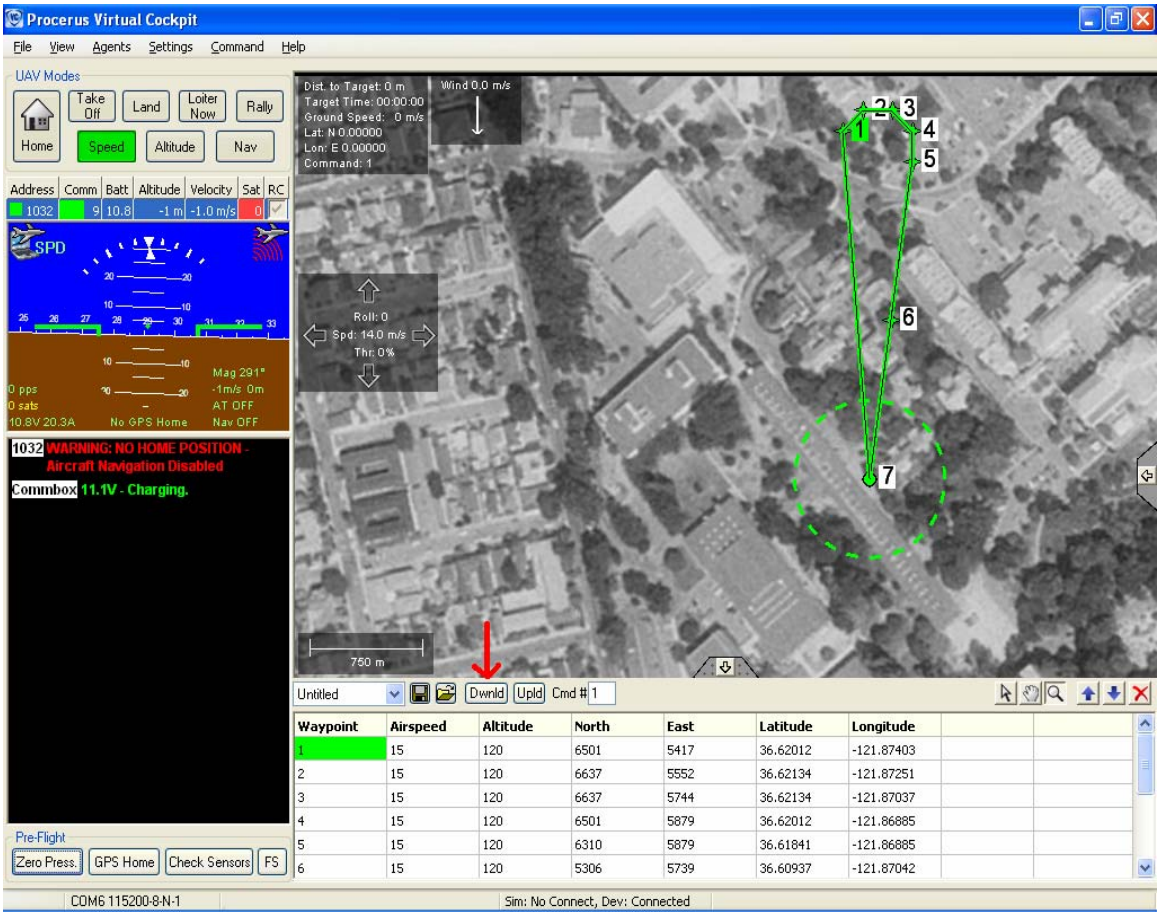


Figure 43. PCPTAv1 flight plan displayed on the VC Geo-referenced Map

IV. TESTING AND RESULTS

A. TESTING/EXPERIMENTATION OVERVIEW

PCPTAv1 must be shown to exhibit the functionality dictated in the previous chapter through rigorous experimentation. The goals set during the development of this project were to enable communications between a user level application and a Kestrel Autopilot, and to create optimal flight plans in two dimensions based on WSN output. A two phased testing approach was used to address these goals.

Phase I presents the reader with evidence supporting PCPTAv1's ability to properly craft Kestrel Autopilot packets and communicate flight plans to the active UAV. While investigating communications, the flight plan produced in each test case is assessed for adherence to the two-dimensional flight optimization scheme detailed in Chapter III. Phase I test cases each involve a WSN investigation scenario, where the active UAV is sent to the WSN cluster reporting the contact. Excluding final destination calculations, the same functions are called to construct flight plans regardless of the selected scenario. WSN investigation scenario flight plans are easier to assess for correctness because the final destination is included in the (user controlled) WSN output, whereas, contact interception scenarios require final destination calculation, which introduces another layer of complexity.

Phase II of the experimentation focuses solely on proving contact interception point calculation accuracy. In each test case, the contact interception scenario is selected. Initial bearing and time to interception are computed by solving a system of non-linear equations and compared to the values produced in PCPTAv1. These are used to generate intercept locations and a distance in meters between PCPTAv1's estimated interception point and the correct location.

B. TESTBED

All experimentation was completed on the Naval Postgraduate School campus using the Procerus Unicorn UAV. The Unicorn was outfitted with a Kestrel Autopilot, Furuno GH-81D GPS unit, and an AeroComm AC4490-1000-M3 RF modem. A Dell Latitude (110L) laptop computer attached to a Procerus Commbox formed the base

station, and both VC and PCPTAv1 (compiled using Microsoft Visual Studio 2005 Professional Edition) were run at the base station computer. Contact data was manipulated by changing (manually) the contents of "contact.txt" to represent the proper WSN output for each case.

C. PHASE I: KESTREL AUTOPILOT PACKET TRANSLATION AND OPTIMIZED FLIGHT PLAN TESTING

1. PCPTAv1/Kestrel Autopilot Communications Testing

To establish communications between PCPTAv1 and a Kestrel Autopilot, properly constructed Packet Forwarding Setup packets must be sent to VC, and command packets crafted within PCPTAv1 must adhere to the Kestrel Autopilot System specification. Each Phase I test case is run using the input specified in Table 9. The flight path output produced by the PathPlanner class will be placed in a text file named "dat.txt" to serve as the input to the packet crafting function (sendPacket). This output is compared to the waypoints viewed in VC after the "download" button is pressed to retrieve the active Kestrel Autopilot's updated flight plan. If they are the same in every test case, one may reasonably assume that the communications functionality within PCPTAv1 is correct.

2. Optimized Flight Plan Testing

Flight paths are optimized in two dimensions by entering the UAV into a turn circle with minimized turn radius until it is headed toward its destination. Then the UAV must fly straight to its destination. PCPTAv1 implements this idea by sending the active UAV on a series of 45 degree turns around a circle with radius equal to the UAV's minimum turn radius. At most, eight waypoints are used to represent a turn. The UAV breaks off of its turn circle when the destination's bearing falls within 45 degrees of the UAV's current heading. If the destination is within a distance twice the turn radius of the active UAV and not within 5 degrees of the UAV's current heading, the UAV is sent forward a distance four times its turn radius before the turn is started.

a. Test Case Orientation

Test case destination points were chosen to elicit a specific response from PCPTAv1. Four sets of cases were fashioned, each containing eleven individual tests. The first set is laid out as seen in Figure 44. The second set has the same orientation, but

with a 90 degree phase shift in the positive direction, making the UAV's heading 135 degrees true. The third and fourth sets apply a 180 and 270 degree phase shift respectively. With reference to Figure 44, cases 2, 4, 5 and 6 should produce flight plans commanding the UAV to fly straight to the destination. Destinations 2 and 5 are directly in front of the UAV, while 4 and 6 require less than a 45 degree turn and are far enough away to allow the Kestrel Autopilot to guide the UAV on a relatively straight path to its destination. Cases 1, 3, 7 and 8 were designed to cause flight plan output sending the UAV forward a distance four times its turn radius, then into a turn and back to its destination. The UAV should make a right turn in cases 3 and 8, and a left in 1 and 7. This behavior is instigated by the proximity of these points. They are within a distance twice the turn radius of the UAV from its current position, and would require more than a 5 degree turn in either direction. The flight plans produced in cases 9 and 10 should result in an instantaneous turn left and right respectively, followed by straight flight to the destination. These destinations are behind the UAV's current position and are far enough away to break into an immediate turn. Case 11 was designed to test PCPTAv1's ability to recognize whether or not a destination is out of range, and should result in a message telling the user that the active UAV is unable to reach the WSN cluster. These responses should hold true for each case set.

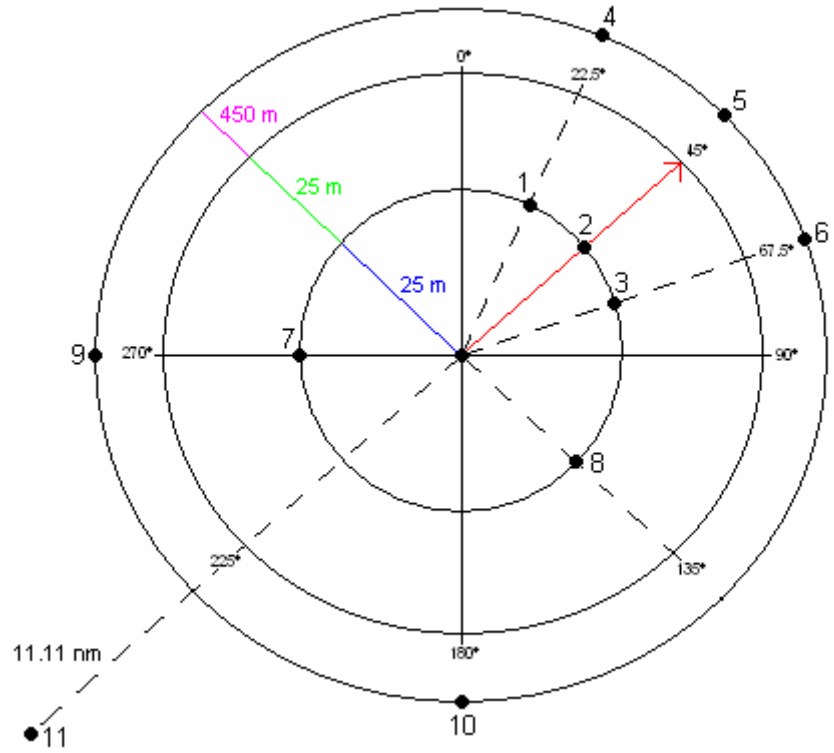


Figure 44. Phase I Test Case Layout (UAV bearing 45 degrees true)

b. Testing Method

In each of the 44 tests, expected behavior of each case is assessed and compared to PCPTAv1's flight plan output. Test cases were designed with a specific output in mind; therefore, it is relatively easy to spot successes and failures. The pattern formed by the waypoints in each flight plan is more important than specific distances and measurements since all of the functions utilized to measure distance and produce new coordinates based on current position and bearing are also used to calculate intercept scenario final destinations. Phase II will test this aspect thoroughly. As all inputs and outputs are provided below, these optimized flight plan tests are highly repeatable.

Test case input is broken down into four sets of 11 tests. Test case numbers correspond to Figure 44, while set numbers define the phase shift applied to the diagram. The following table describes the testing inputs used:

Set/ Test Case		UAV Position	UAV Initial Bearing True North=0.0 True East=90.0 (degrees)	UAV Turn Radius (m)	WSN Cluster Location
1	1	36.59560 -121.87613	45.0	50	36.595810 -121.87602
1	2	36.59560 -121.87613	45.0	50	36.595757 -121.87593
1	3	36.59560 -121.87613	45.0	50	36.595688 -121.87588
1	4	36.59560 -121.87613	45.0	50	36.599773 -121.87403
1	5	36.59560 -121.87613	45.0	50	36.598782 -121.87217
1	6	36.59560 -121.87613	45.0	50	36.597355 -121.87097
1	7	36.59560 -121.87613	45.0	50	36.595600 -121.87642
1	8	36.59560 -121.87613	45.0	50	36.595516 -121.87588
1	9	36.59560 -121.87613	45.0	50	36.595600 -121.88174
1	10	36.59560 -121.87613	45.0	50	36.591099 -121.87614
1	11	36.59560 -121.87613	45.0	50	36.524857 -121.96412
2	1	36.59968 -121.87660	135.0	50	36.599598 -121.87634
2	2	36.59968 -121.87660	135.0	50	36.599525 -121.87640
2	3	36.59968 -121.87660	135.0	50	36.599472 -121.87649
2	4	36.59968 -121.87660	135.0	50	36.598000 -121.87141
2	5	36.59968 -121.87660	135.0	50	36.596500 -121.87263

2	6	36.59968 -121.87660	135.0	50	36.595539 -121.87441
2	7	36.59968 -121.87660	135.0	50	36.599911 -121.87659
2	8	36.59968 -121.87660	135.0	50	36.599472 -121.87671
2	9	36.59968 -121.87660	135.0	50	36.604183 -121.87659
2	10	36.59968 -121.87660	135.0	50	36.599682 -121.88220
2	11	36.59968 -121.87660	135.0	50	36.528942 -121.78860
3	1	36.60078 -121.88184	225.0	50	36.600574 -121.88195
3	2	36.60078 -121.88184	225.0	50	36.600624 -121.88205
3	3	36.60078 -121.88184	225.0	50	36.600693 -121.88210
3	4	36.60078 -121.88184	225.0	50	36.596607 -121.88395
3	5	36.60078 -121.88184	225.0	50	36.597599 -121.88580
3	6	36.60078 -121.88184	225.0	50	36.599022 -121.88701
3	7	36.60078 -121.88184	225.0	50	36.600780 -121.88155
3	8	36.60078 -121.88184	225.0	50	36.600868 -121.88210
3	9	36.60078 -121.88184	225.0	50	36.600780 -121.87624
3	10	36.60078 -121.88184	225.0	50	36.605282 -121.88184
3	11	36.60078 -121.88184	225.0	50	36.671455 -121.79368
4	1	36.60214 -121.87402	315.0	50	36.602222 -121.87429

4	2	36.60214 -121.87402	315.0	50	36.602295 -121.87424
4	3	36.60214 -121.87402	315.0	50	36.602345 -121.87414
4	4	36.60214 -121.87402	315.0	50	36.603821 -121.87922
4	5	36.60214 -121.87402	315.0	50	36.605320 -121.87799
4	6	36.60214 -121.87402	315.0	50	36.606281 -121.87621
4	7	36.60214 -121.87402	315.0	50	36.601910 -121.87403
4	8	36.60214 -121.87402	315.0	50	36.602295 -121.87383
4	9	36.60214 -121.87402	315.0	50	36.597637 -121.87403
4	10	36.60214 -121.87402	315.0	50	36.602139 -121.86843
4	11	36.60214 -121.87402	315.0	50	36.672817 -121.96220

Table 9. Phase I Test Inputs

3. Phase I Results and Analysis

The results of Phase I testing can be viewed in Table 10. Test cases that both adhere to the two-dimensional optimized flight scheme described in the previous chapter, and are received and understood by the active Kestrel Autopilot are deemed successes. Those failing on either aspect of the test are considered failures.

a. PCPTAv1 Computed Flight Plan Output Format

PCPTAv1 output was copied and pasted from "dat.txt" upon each execution. The data is in a format understood by the Kestrel Autopilot, and therefore requires some explanation. The first number on each row describes the type of command being sent. PCPTAv1 uses only Goto (2) and Loiter (4) command packets to construct flight plans. Goto commands are used to turn the active UAV onto a heading equal to the destination bearing, while Loiter commands are issued to send the UAV to its final

destination. The last command in every flight plan will always be a loiter. The second number in each PCPTAv1 flight plan output line describes the altitude at which UAV is instructed to fly. The third details its speed. A constant multiplier is applied to both units, per the Kestrel Autopilot System specification. Altitude units are "meters * 10," while velocity units are "2 * meters/second." Goto command lines follow with the latitude and longitude (degrees) of the waypoint, and finish with a payload value (0). Loiter commands proceed somewhat differently, as they require additional information. Following the velocity value is the length of time (seconds) the UAV should remain in a circular holding pattern at the indicated waypoint. A zero in this location tells the UAV to remain onsite for as long as possible (until power resources are nearly depleted). The next number details the loiter circle radius (meters), which is derived from the UAV turn radius inputted by the user. Loiter commands finish in the same manner as Goto lines, with waypoint coordinates and an empty payload value.

b. VC Flight Plan Output Format

Flight plans were viewed in VC upon each test case execution. The observed flight plans are formatted in Table 10 in a very similar fashion to the PCPTAv1 output in the opposing column. Loiter command lines are identified with an "L," while Goto commands are begun with a "G." The VC display factors out the multipliers applied to various command values, however, the data is ordered in the same manner. VC rounds longitudes and latitudes within its display to a precision of five decimal places, which explains the truncation of the observed values. Additionally, the payload is not displayed within the VC GUI.

Set/ Test Case		**PCPTAv1 Computed Flight Plan**	**Flight Plan Viewed in VC**	**Success**
1	1	2 1200 30 36.59687042 -121.8745499 0 2 1200 30 36.59704208 -121.8745499 0 2 1200 30 36.59716415 -121.8747025 0 2 1200 30 36.59716415 -121.8749161 0 2 1200 30 36.59704208 -121.8750687 0 2 1200 30 36.59642792 -121.8755493 0 4 1200 30 0 50 36.59580994 -121.8760223 0	G 120 15 36.59687 -121.87455 G 120 15 36.59704 -121.87455 G 120 15 36.59716 -121.87470 G 120 15 36.59716 -121.87492 G 120 15 36.59704 -121.87507 G 120 15 36.59643 -121.87555 L 120 15 0 50 36.59581 -121.87602	Yes
1	2	4 1200 30 0 50 36.59575653 -121.8759308 0	L 120 15 0 50 36.59576 -121.87593	Yes
1	3	2 1200 30 36.59687042 -121.8745499 0 2 1200 30 36.59687042 -121.8743286 0	G 120 15 36.59687 -121.87455 G 120 15 36.59687 -121.87433	Yes

		2 1200 30 36.59674835 -121.8741837 0 2 1200 30 36.59657669 -121.8741837 0 2 1200 30 36.59645462 -121.8743286 0 2 1200 30 36.59607315 -121.8751068 0 4 1200 30 0 50 36.59568787 -121.8758774 0	G 120 15 36.59675 -121.87418 G 120 15 36.59658 -121.87418 G 120 15 36.59645 -121.87433 G 120 15 36.59607 -121.87511 L 120 15 0 50 36.59569 -121.87588	
1	4	4 1200 30 0 50 36.59977341 -121.8740311 0	L 120 15 0 50 36.59977 -121.87403	Yes
1	5	4 1200 30 0 50 36.59878159 -121.8721695 0	L 120 15 0 50 36.59878 -121.87217	Yes
1	6	4 1200 30 0 50 36.59735489 -121.8709717 0	L 120 15 0 50 36.59735 -121.87097	Yes
1	7	2 1200 30 36.59687042 -121.8745499 0 2 1200 30 36.59704208 -121.8745499 0 2 1200 30 36.59716415 -121.8747025 0 2 1200 30 36.59716415 -121.8749161 0 2 1200 30 36.59704208 -121.8750687 0 2 1200 30 36.59631729 -121.8757477 0 4 1200 30 0 50 36.59560013 -121.8764191 0	G 120 15 36.59687 -121.87455 G 120 15 36.59704 -121.87455 G 120 15 36.59716 -121.87470 G 120 15 36.59716 -121.87492 G 120 15 36.59704 -121.87507 G 120 15 36.59632 -121.87575 L 120 15 0 50 36.59560 -121.87642	Yes
1	8	2 1200 30 36.59687042 -121.8745499 0 2 1200 30 36.59687042 -121.8743286 0 2 1200 30 36.59674835 -121.8741837 0 2 1200 30 36.59657669 -121.8741837 0 2 1200 30 36.59645462 -121.8743286 0 2 1200 30 36.59598541 -121.8751068 0 4 1200 30 0 50 36.5955162 -121.8758774 0	G 120 15 36.59687 -121.87455 G 120 15 36.59687 -121.87433 G 120 15 36.59675 -121.87418 G 120 15 36.59658 -121.87418 G 120 15 36.59645 -121.87433 G 120 15 36.59599 -121.87511 L 120 15 0 50 36.5955 -121.87588	Yes
1	9	2 1200 30 36.59577179 -121.8761368 0 2 1200 30 36.59589386 -121.8762817 0 2 1200 30 36.59589386 -121.876503 0 2 1200 30 36.5957489 -121.8791275 0 4 1200 30 0 50 36.59560013 -121.8817368 0	G 120 15 36.59577 -121.87614 G 120 15 36.59589 -121.87628 G 120 15 36.59589 -121.87650 G 120 15 36.59575 -121.87913 L 120 15 0 50 36.59560 -121.88174	Yes
1	10	2 1200 30 36.59560013 -121.8759155 0 2 1200 30 36.59547806 -121.8757629 0 2 1200 30 36.5953064 -121.8757629 0 2 1200 30 36.5932045 -121.8759537 0 4 1200 30 0 50 36.59109879 -121.8761368 0	G 120 15 36.59560 -121.87592 G 120 15 36.59548 -121.87576 G 120 15 36.59531 -121.87576 G 120 15 36.59320 -121.87595 L 120 15 0 50 36.59110 -121.87614	Yes
1	11	The destination is out of range	None	Yes
2	1	2 1200 30 36.59841156 -121.8750153 0 2 1200 30 36.59841156 -121.874794 0 2 1200 30 36.59853363 -121.8746414 0 2 1200 30 36.59870529 -121.8746414 0 2 1200 30 36.59882736 -121.874794 0 2 1200 30 36.59921265 -121.8755569 0 4 1200 30 0 50 36.59959793 -121.8763428 0	G 120 15 36.59841 -121.87502 G 120 15 36.59841 -121.87479 G 120 15 36.59853 -121.87464 G 120 15 36.59871 -121.87464 G 120 15 36.59883 -121.87479 G 120 15 36.59921 -121.87556 L 120 15 0 50 36.59960 -121.87634	Yes
2	2	4 1200 30 0 50 36.59952545 -121.8764038 0	L 120 15 0 50 36.59953 -121.87640	Yes
2	3	2 1200 30 36.59841156 -121.8750153 0 2 1200 30 36.5982399 -121.8750153 0 2 1200 30 36.59811783 -121.8751602 0 2 1200 30 36.59811783 -121.8753815 0 2 1200 30 36.5982399 -121.8755341 0 2 1200 30 36.59885788 -121.8760071 0 4 1200 30 0 50 36.59947205 -121.8764877 0	G 120 15 36.59841 -121.87502 G 120 15 36.59829 -121.87502 G 120 15 36.59812 -121.87516 G 120 15 36.59812 -121.87538 G 120 15 36.59824 -121.87553 G 120 15 36.59886 -121.87601 L 120 15 0 50 36.59947 -121.87649	Yes
2	4	4 1200 30 0 50 36.59799957 -121.8714066 0	L 120 15 0 50 36.59800 -121.87141	Yes
2	5	4 1200 30 0 50 36.5965004 -121.8726273 0	L 120 15 0 50 36.59650 -121.87263	Yes

2	6	4 1200 30 0 50 36.59553909 -121.8744125 0	L 120 15 0 50 36.59554 -121.87441	Yes
2	7	2 1200 30 36.59841156 -121.8750153 0 2 1200 30 36.59841156 -121.874794 0 2 1200 30 36.59853363 -121.8746414 0 2 1200 30 36.59870529 -121.8746414 0 2 1200 30 36.59882736 -121.874794 0 2 1200 30 36.59936905 -121.8756943 0 4 1200 30 0 50 36.59991074 -121.8765869 0	G 120 15 36.59841 -121.87502 G 120 15 36.59841 -121.87479 G 120 15 36.59853 -121.87464 G 120 15 36.59871 -121.87464 G 120 15 36.59883 -121.87479 G 120 15 36.59937 -121.87569 L 120 15 0 50 36.59991 -121.87659	Yes
2	8	2 1200 30 36.59841156 -121.8750153 0 2 1200 30 36.5982399 -121.8750153 0 2 1200 30 36.59811783 -121.8751602 0 2 1200 30 36.59811783 -121.8753815 0 2 1200 30 36.5982399 -121.8755341 0 2 1200 30 36.59885788 -121.8761215 0 4 1200 30 0 50 36.59947205 -121.876709 0	G 120 15 36.59841 -121.87502 G 120 15 36.59824 -121.87502 G 120 15 36.59812 -121.87516 G 120 15 36.59812 -121.87538 G 120 15 36.59824 -121.87553 G 120 15 36.59886 -121.87612 L 120 15 0 50 36.59947 -121.87671	Yes
2	9	2 1200 30 36.59968185 -121.8763809 0 2 1200 30 36.59980774 -121.8762283 0 2 1200 30 36.59997559 -121.8762283 0 2 1200 30 36.60207748 -121.8764038 0 4 1200 30 0 50 36.6041832 -121.8765869 0	G 120 15 36.59968 -121.87638 G 120 15 36.59981 -121.87623 G 120 15 36.59998 -121.87623 G 120 15 36.60208 -121.87640 L 120 15 0 50 36.60418 -121.87659	Yes
2	10	2 1200 30 36.59951401 -121.8765945 0 2 1200 30 36.59939194 -121.8767471 0 2 1200 30 36.59939194 -121.8769684 0 2 1200 30 36.5995369 -121.8795929 0 4 1200 30 0 50 36.59968185 -121.8822021 0	G 120 15 36.59951 -121.87659 G 120 15 36.59939 -121.87675 G 120 15 36.59939 -121.87697 G 120 15 36.59954 -121.87959 L 120 15 0 50 36.59968 -121.88220	Yes
2	11	The destination is out of range	None	Yes
3	1	2 1200 30 36.59950638 -121.8834305 0 2 1200 30 36.59933853 -121.8834305 0 2 1200 30 36.59921265 -121.8832779 0 2 1200 30 36.59921265 -121.8830566 0 2 1200 30 36.59933853 -121.8829117 0 2 1200 30 36.59996033 -121.882431 0 4 1200 30 0 50 36.60057449 -121.8819504 0	G 120 15 36.59951 -121.88343 G 120 15 36.59934 -121.88343 G 120 15 36.59921 -121.88328 G 120 15 36.59921 -121.88306 G 120 15 36.59934 -121.88291 G 120 15 36.59996 -121.88243 L 120 15 0 50 36.60057 -121.88195	Yes
3	2	4 1200 30 0 50 36.60062408 -121.8820496 0	L 120 15 0 50 36.60062 -121.88205	Yes
3	3	2 1200 30 36.59950638 -121.8834305 0 2 1200 30 36.59950638 -121.8836441 0 2 1200 30 36.59963226 -121.8837967 0 2 1200 30 36.59980011 -121.8837967 0 2 1200 30 36.59992218 -121.8836441 0 2 1200 30 36.60031128 -121.8828812 0 4 1200 30 0 50 36.60069275 -121.882103 0	G 120 15 36.59951 -121.88343 G 120 15 36.59951 -121.88364 G 120 15 36.59963 -121.88380 G 120 15 36.59980 -121.88380 G 120 15 36.59992 -121.88364 G 120 15 36.60031 -121.88288 L 120 15 0 50 36.60069 -121.8821	Yes
3	4	4 1200 30 0 50 36.59660721 -121.8839493 0	L 120 15 0 50 36.59661 -121.88395	Yes
3	5	4 1200 30 0 50 36.59759903 -121.8858032 0	L 120 15 0 50 36.59760 -121.88580	Yes
3	6	4 1200 30 0 50 36.59902191 -121.8870087 0	L 120 15 0 50 36.59902 -121.88701	Yes
3	7	2 1200 30 36.59950638 -121.8834305 0 2 1200 30 36.59933853 -121.8834305 0 2 1200 30 36.59921265 -121.8832779 0 2 1200 30 36.59921265 -121.8830566 0 2 1200 30 36.59933853 -121.8829117 0 2 1200 30 36.60006332 -121.8822403 0 4 1200 30 0 50 36.60078049 -121.8815536 0	G 120 15 36.59951 -121.88343 G 120 15 36.59934 -121.88343 G 120 15 36.59921 -121.88328 G 120 15 36.59921 -121.88306 G 120 15 36.59934 -121.88291 G 120 15 36.60006 -121.88224 L 120 15 0 50 36.60078 -121.88155	Yes
3	8	2 1200 30 36.59950638 -121.8834305 0	G 120 15 36.59951 -121.88343	Yes

		2 1200 30 36.59950638 -121.8836441 0 2 1200 30 36.59963226 -121.8837967 0 2 1200 30 36.59980011 -121.8837967 0 2 1200 30 36.59992218 -121.8836441 0 2 1200 30 36.60039902 -121.8828812 0 4 1200 30 0 50 36.60086823 -121.882103 0	G 120 15 36.59951 -121.88364 G 120 15 36.59963 -121.88380 G 120 15 36.59980 -121.88380 G 120 15 36.59992 -121.88364 G 120 15 36.60040 -121.88288 L 120 15 0 50 36.60087 -121.88210	
3	9	2 1200 30 36.60060883 -121.8818436 0 2 1200 30 36.60048676 -121.881691 0 2 1200 30 36.60048676 -121.8814774 0 2 1200 30 36.60063553 -121.8788528 0 4 1200 30 0 50 36.60078049 -121.8762436 0	G 120 15 36.60061 -121.88184 G 120 15 36.60049 -121.88169 G 120 15 36.60049 -121.88148 G 120 15 36.60064 -121.87885 L 120 15 0 50 36.60078 -121.87624	Yes
3	10	2 1200 30 36.60078049 -121.8820648 0 2 1200 30 36.60090256 -121.8822098 0 2 1200 30 36.60107422 -121.8822098 0 2 1200 30 36.60317993 -121.8820343 0 4 1200 30 0 50 36.60528183 -121.8818436 0	G 120 15 36.60078 -121.88206 G 120 15 36.60090 -121.88221 G 120 15 36.60107 -121.88221 G 120 15 36.60318 -121.88203 L 120 15 0 50 36.60528 -121.88184	Yes
3	11	The destination is out of range.	None	Yes
4	1	2 1200 30 36.60340881 -121.875618 0 2 1200 30 36.603405 -121.8758316 0 2 1200 30 36.60327911 -121.8759842 0 2 1200 30 36.60310745 -121.8759842 0 2 1200 30 36.60298538 -121.8758316 0 2 1200 30 36.60260391 -121.8750534 0 4 1200 30 0 50 36.60222244 -121.8742905 0	G 120 15 36.60341 -121.87562 G 120 15 36.60341 -121.87583 G 120 15 36.60328 -121.87598 G 120 15 36.60311 -121.87598 G 120 15 36.60299 -121.87583 G 120 15 36.60260 -121.87505 L 120 15 0 50 36.60222 -121.87429	Yes
4	2	4 1200 30 0 50 36.60229492 -121.8742371 0	L 120 15 50 36.60229 -121.87424	Yes
4	3	2 1200 30 36.60340881 -121.875618 0 2 1200 30 36.60357285 -121.875618 0 2 1200 30 36.60369492 -121.8754654 0 2 1200 30 36.60369492 -121.8752441 0 2 1200 30 36.60357285 -121.8750916 0 2 1200 30 36.60295486 -121.8746185 0 4 1200 30 0 50 36.60234451 -121.8741379 0	G 120 15 36.60341 -121.87562 G 120 15 36.60357 -121.87562 G 120 15 36.60369 -121.87547 G 120 15 36.60369 -121.87524 G 120 15 36.60357 -121.87509 G 120 15 36.60295 -121.87462 L 120 15 0 50 36.60234 -121.87414	Yes
4	4	4 1200 30 0 50 36.6038208 -121.8792191 0	L 120 15 0 50 36.60382 -121.87922	Yes
4	5	4 1200 30 0 50 36.60531998 -121.8779907 0	L 120 15 0 50 36.60532 -121.87799	Yes
4	6	4 1200 30 0 50 36.60628128 -121.8762131 0	L 120 15 0 50 36.60628 -121.87621	Yes
4	7	2 1200 30 36.60340881 -121.875618 0 2 1200 30 36.603405 -121.8758316 0 2 1200 30 36.60327911 -121.8759842 0 2 1200 30 36.60310745 -121.8759842 0 2 1200 30 36.60298538 -121.8758316 0 2 1200 30 36.6024437 -121.8749313 0 4 1200 30 0 50 36.60190964 -121.8740311 0	G 120 15 36.60341 -121.87562 G 120 15 36.60341 -121.87583 G 120 15 36.60328 -121.87598 G 120 15 36.60311 -121.87598 G 120 15 36.60299 -121.87583 G 120 15 36.60244 -121.87493 L 120 15 0 50 36.60191 -121.87403	Yes
4	8	2 1200 30 36.60340881 -121.875618 0 2 1200 30 36.60357285 -121.875618 0 2 1200 30 36.60369492 -121.8754654 0 2 1200 30 36.60369492 -121.8752441 0 2 1200 30 36.60357285 -121.8750916 0 2 1200 30 36.60293198 -121.8744659 0 4 1200 30 0 50 36.60229492 -121.8738327 0	G 120 15 36.60341 -121.87562 G 120 15 36.60357 -121.87562 G 120 15 36.60369 -121.87547 G 120 15 36.60369 -121.87524 G 120 15 36.60357 -121.87509 G 120 15 36.60293 -121.87447 L 120 15 0 50 36.60229 -121.87383	Yes
4	9	2 1200 30 36.60213852 -121.8742447 0 2 1200 30 36.60201263 -121.8743973 0 2 1200 30 36.60184479 -121.8743973 0 2 1200 30 36.59973907 -121.8742065 0	G 120 15 36.60214 -121.87424 G 120 15 36.60201 -121.87440 G 120 15 36.60184 -121.87440 G 120 15 36.59974 -121.87421	Yes

		4 1200 30 0 50 36.59763718 -121.8740311 0	L 120 15 0 50 36.59764 -121.87403	
4	10	2 1200 30 36.60230637 -121.8740311 0 2 1200 30 36.60243225 -121.8738785 0 2 1200 30 36.60243225 -121.8736572 0 2 1200 30 36.60228348 -121.8710556 0 4 1200 30 0 50 36.60213852 -121.8684311 0	G 120 15 36.60231 -121.87403 G 120 15 36.60243 -121.87388 G 120 15 36.60243 -121.87366 G 120 15 36.60228 -121.87106 L 120 15 0 50 36.60214 -121.86843	Yes
4	11	The destination is out of range	None	Yes

Table 10. Phase I Test Results

c. Analysis of Phase I Results

All 44 test cases returned positive results. Every flight plan produced during Phase I testing was optimized in two-dimensions. In each set, tests 2, 4, 5 and 6 produced flight plans directing the UAV to fly straight to its destination, as was expected. Tests 1, 3, 7 and 8 resulted in a series of waypoints taking the UAV 200 meters (four times its turn radius) forward, then making 45 degree turns around a turn circle with radius equal to 50 meters towards its destination. All turns were called in the correct direction. Left turns were consistently made in tests 1, 7 and 9 while right turns were made in the 3rd, 8th and 10th tests of each set. Test 9 and 10 executions returned proper flight plans entailing an immediate turn towards the destination, and PCPTAv1 correctly identified that all test 11 destinations were out of range.

Flight plans were also packetized and communicated to the Kestrel Autopilot correctly. PCPTAv1 dat.txt output was identical in each case to the flight plan observed in VC.

D. PHASE II: CONTACT INTERCEPTION SCENARIO TESTING

1. Contact Interception Calculation Testing

Interception scenarios involve many mathematical processes that are not used during path creation. Intercept location calculations are performed prior to flight plan creation, as no flight plan can be constructed without an established final destination. The accuracy of the initial intercept locations produced by PCPTAv1 is evaluated during Phase II testing.

a. Test Case Orientation

Test case destinations were placed 500 meters from the UAV at relative bearings of 45, 135, 225 and 315 degrees (see Figure 45). UAV heading was maintained

at 0 degrees true through the entire phase. Four tests were conducted at each bearing. The first test involves a contact moving away from the UAV in both X and Y, (with reference to Figure 45) at a greater velocity than the maximum speed of the UAV. This should always cause PCPTAv1 to output a "destination unreachable" message to the user. In the second, the contact is moving away from the UAV in X and Y, but is proceeding at a slower rate than the UAV's maximum speed. The third involves a contact moving closer to the UAV in X and/or Y, with velocity greater than that which the UAV is capable. In the fourth, the contact is moving closer to the UAV in X and/or Y, with a velocity that is less than the UAV's maximum speed. This test should always produce a reachable intercept location.

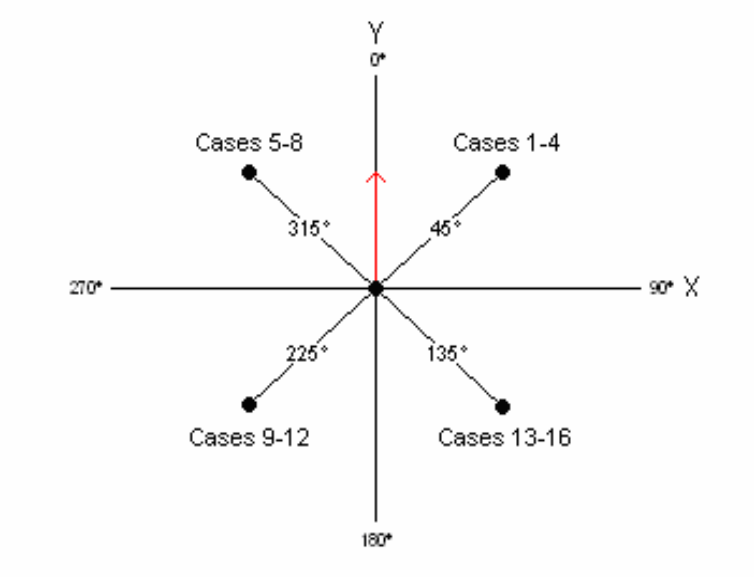


Figure 45. Phase II Test Case Layout (UAV bearing 0 degrees true)

b. Testing Method

PCPTAv1 output will be compared to a mathematical contact interception model. This model is derived in the following manner. At time *t*, the vector "C," describing the contact's position in X and Y (with reference to Figure 45) can be described by: $\vec{C} = (x_0 + ut, y_0 + vt)$, where x_0 is the X component of the contact's initial position, and y_0 is the Y component. The contact moves with velocity *u* in the X direction and *v* in the Y direction. At an angle $\emptyset_C$ and velocity *w*, $u=w*\cos(\emptyset_C)$ and $v=w*\sin(\emptyset_C)$. In this model, the UAV's initial position is (0,0), its velocity is *p* and it

must travel at a heading $Ø_I$ to get to the intercept location. UAV position $\vec{U}$ can be described by: $\vec{U}$ = (pt*cos($Ø_I$), pt*sin($Ø_I$)). To intercept, the contact and UAV's positions must be equal in X and Y; therefore the interception point is: ($x_0 + ut$ = pt*cos($Ø_I$), $y_0 + vt = pt$*sin($Ø_I$)). These equations can be reorganized to:

$$0 = x_0 + wt^*\cos(Ø_C) - pt^*\cos(Ø_I)$$

$$0 = y_0 + wt^*\sin(Ø_C) - pt^*\sin(Ø_I)$$

Figure 46. System of nonlinear equations describing active UAV/contact interception points

We are left with two equations and two unknowns (t and $Ø_I$), a solvable system. Input and output angles in these equations are based on a coordinate system where true north is equal to 90 degrees and true east is at 0 degrees. The time and angle of interception computed with this system correspond to the initial time of interception and initial UAV heading, as this calculation does not factor in the time it would take the UAV to turn toward its destination. Initial destination values provide a sufficient test to PCPTAv1's interception scenario output because the computation responsible for determining the final destination calls the same functions in the same order as the initial calculation. Also, the distance input used to finalize the destination is based on the circumference of the UAV's turn circle; a very basic calculation given the fact that the UAV turn radius is inputted by the user.

Phase II testing was conducted by comparing PCPTAv1 initial intercept values to those solved for using the system of equations described above. PCPTAv1 does not output these numbers, but they are easily attained during execution with the Microsoft Visual Studio 2005 Professional Edition debugger. The mathematical model was solved for each test case using Maple 10. An unabridged copy of the Maple 10 input and output for each test case can be viewed in Appendix B. UAV inputs for each case are detailed in Table 11, WSN contact inputs in Table 12. Once t and $Ø_I$ are acquired in PCPTAv1 and Maple 10, they are used as inputs, along with contact velocity, to the

equations described in Figure 36 to find a latitude and longitude for each calculated interception point. The distance between these points is then measured using the haversine formula.

Test Case #	**UAV Initial Position**	**UAV Turn Radius** (m)	**UAV Range** (nm)	**UAV Max Speed** (m/s)	**UAV Initial Heading** True North=0.0 True East=90.0 (degrees)
1	36.59560 -121.87613	50.0	5.0	15.0	0.0
2	36.59560 -121.87613	50.0	5.0	15.0	0.0
3	36.59560 -121.87613	50.0	5.0	15.0	0.0
4	36.59560 -121.87613	50.0	5.0	15.0	0.0
5	36.59968 -121.87660	50.0	5.0	15.0	0.0
6	36.59968 -121.87660	50.0	5.0	15.0	0.0
7	36.59968 -121.87660	50.0	5.0	15.0	0.0
8	36.59968 -121.87660	50.0	5.0	15.0	0.0
9	36.60078 -121.88184	50.0	5.0	20.0	0.0
10	36.60078 -121.88184	50.0	5.0	20.0	0.0
11	36.60078 -121.88184	50.0	5.0	20.0	0.0
12	36.60078 -121.88184	50.0	5.0	20.0	0.0
13	36.60214 -121.87402	50.0	5.0	30.0	0.0
14	36.60214 -121.87402	50.0	5.0	30.0	0.0
15	36.60214 -121.87402	50.0	5.0	30.0	0.0
16	36.60214 -121.87402	50.0	5.0	30.0	0.0

Table 11. Phase II Test Case Inputs: Active UAV

Test Case #	**Contact Initial Position** (WSN Location)	**Contact Heading** True North=0.0 True East=90.0 (degrees)	**Contact Heading** True North=90.0 True East=0.0 (degrees)	**Contact Speed** (m/s)
1	36.598778985112 -121.87217303657	45.0	45.0	20.0
2	36.598778985112 -121.87217303657	23.0	67.0	5.0
3	36.598778985112 -121.87217303657	250.0	200.0	25.0
4	36.598778985112 -121.87217303657	260.0	190.0	10.0
5	36.5965034321 -121.87263760624	160.0	290.0	16.0
6	36.5965034321 -121.87263760624	110.0	340.0	14.0
7	36.5965034321 -121.87263760624	305.0	145.0	30.0
8	36.5965034321 -121.87263760624	1.0	89.0	2.0
9	36.5976009380 -121.88580394110	200.0	250.0	100.0
10	36.5976009380 -121.88580394110	265.0	185.0	19.0
11	36.5976009380 -121.88580394110	85.0	5.0	50.0
12	36.5976009380 -121.88580394110	135.0	315.0	15.0
13	36.60531711265 -121.87799060184	270.0	180.0	32.0
14	36.60531711265 -121.87799060184	335.0	95.0	25.0
15	36.60531711265 -121.87799060184	170.0	280.0	35.0
16	36.60531711265 -121.87799060184	83.0	7.0	13.0

Table 12. Phase II Test Case Inputs: WSN Output

2. Phase II Results and Analysis

Maple 10 calculations yielded five cases where the time to intercept was negative. A negative time value, such as -14.2857233 in test case 1, means the UAV would require

14.2857233 additional seconds to intercept the contact. In other words, this indicates a contact is impossible for the active UAV to intercept. PCPTAv1 correctly reported the destination out of range in each of these cases.

Phase II results show PCPTAv1's contact interception point estimations to be extremely accurate. According to the measurement between calculated and estimated intercept locations, the largest divergence was approximately 0.0156 meters. This is somewhat disconcerting, as PCPTAv1 theoretically should only be accurate to within five meters. The PCPTAv1 and Maple 10 initial time to intercept and initial bearing values are very close, but contain enough disparity to result in greater locational differences than those reported using the testing scheme outlined in the previous section. The (nextPoint) function used to acquire the interception latitudes and longitudes based on calculated initial values clearly lacks the accuracy required to reflect small differences in heading and distance in its output.

To address this shortcoming, an additional column was added to Table 14, detailing the results of an additional comparison. The initial time to intercept can be converted to the distance traveled by the contact since its velocity is known. The fact that both the calculated and estimated interception point must fall along the contact's projected path enables us to assess the variance in the contacts position between these locations using the difference in time to initial intercept. For example, in test case 2, taking 50.0-48.79618653 yields 1.20381347 seconds. Multiplied by a contact velocity of 5 m/s, we see that the estimated interception point produced by PCPTAv1 is 6.01906735 meters away from the correct location.

This evaluation method returned realistic results. PCPTAv1 output was within 5 meters of the actual intercept location in seven of the sixteen test cases. Only in the fourteenth test did PCPTAv1 produce a location that was inaccurate by more than approximately 8 meters. The error in this case can be attributed to rounding and the imprecision of the nextPoint function (Figure 36), as evinced by the Phase II test results. The time to intercept computed within PCPTAv1 was never off by more than 1 second, and the worst initial bearing error recorded was 1.345 degrees.

Since the time differential is multiplied by contact velocity, it seems logical to assume faster contacts will result in greater intercept location error. This however did not prove true. Tests involving a contact with velocity greater than UAV max speed had an average error of 6.2888 meters, while those involving a slower contact had error averaging 0.9931 meters. Cases where the contact was moving closer to the UAV in X or Y, at a higher velocity were consistently the most accurate, while those involving a contact moving away from the UAV in both X and Y at a slower velocity recorded the worst error. It is not currently understood why this relationship exists.

Phase II test results showed that PCPTAv1 produces intercept locations with greater error than five meters quite regularly. Phase II exposed the imprecision of the nextPoint function over small distances, which is the probable source of intercept location error within PCPTAv1 calculations. This error is acceptable in the context of surveillance operations, but must be noted and addressed during the development of this project. Although PCPTAv1 intercept scenario calculations exhibit an error above that expected from the estimation scheme, Phase II testing showed the application is capable of successfully guiding UAV—WSN contact interception operations.

	PCPTAv1 Results		**Maple 10 Results**		
Test Case #	**Initial Time to Intercept** (s)	**Initial Bearing** True North=0.0 True East=90.0 (degrees)	**Initial Time to Intercept** (s)	**Initial Bearing** True North=90.0 True East=0.0 (radians)	**Initial Bearing** True North=0.0 True East=90.0 (degrees)
1	Destination out of range.		-14.2857233 (None)	-2.356194490 (None)	N/A
2	50.000000	37.653172	48.79618653	0.9105929386	37.82686776
3	14.999999	359.05942	15.01263570	1.566925476	0.2217834137
4	23.000000	21.320089	22.67412202	1.177773695	22.51853805
5	Destination out of range.		-17.9273490 (None)	2.823847404 (None)	N/A
6	456.78571	111.76167	456.4175971	-.3799353884	111.7686942
7	11.499999	156.22878	11.46511546	-1.140085211	155.3220709
8	32.500000	128.60730	30.63694499	-.6893385368	129.4961888
9	Destination out of range.		-4.72035089 + 1.93905578 I (None)	-.7853981634 + 1.379921070 I (None)	N/A
10	389.47368	262.63806	389.6547793	-3.013073433	262.6363911
11	Destination out of range.		9.119574378 +	-.7853981634 +	N/A

			5.99006596 I (None)	1.052527834 I (None)	
12	38.333333	175.79776	37.79646228	-1.508132060	176.409602
13	Destination out of range.		-11.8133120 (None)	-1.639905048 (None)	N/A
14	81.600000	347.46674	80.88048722	1.790909649	347.3884356
15	9.8571428	271.69440	9.810937985	3.089289635	272.9967422
16	13.846153	335.73236	13.64065931	2.017821518	334.3873432

Table 13. Phase II Test Results: Data Collection

Test Case #	**PCPTAv1 Initial Interception Point**	**Maple 10 Initial Interception Point**	**Distance Between Interception Points** (m)	**Gap Between Estimated and Calculated Distance Traveled By Contact** (m)
1	Destination out of range.			
2	36.595602883422572 -121.8761272289175	36.595602807405356 -121.8761272850218	0.0098186221061803327	6.01906735
3	36.595601092440269 -121.8761300223393	36.595601093499738 -121.8761299947279	0.0024661339731486824	0.31589253
4	36.595601560649712 -121.8761292413374	36.595601525641833 -121.8761292121698	0.0046801626935933007	3.2587798
5	Destination out of range.			
6	36.599667664552555 -121.8765615097983	36.599667670708278 -121.8765615426991	0.0030137181618778179	5.1535806
7	36.599679233413973 -121.8765995794269	36.599679241161816 -121.8765995656908	0.0014975960668915573	1.0465362
8	36.599678522870910 -121.8765976957696	36.599678580661660 -121.8765978550277	0.015591235880672103	3.7261102
9	Destination out of range.			
10	36.600775153182695 -121.8818867277044	36.600775149835798 -121.8818867492561	0.0019582280519072517	3.4408867
11	Destination out of range.			
12	36.600776287117647 -121.8818396601896	36.600776336454182 -121.8818397136625	0.0072670764261777553	8.0530658
13	Destination out of range.			
14	36.602151604079403 -121.8740232133115	36.602151498254599 -121.8740232045554	0.011785165552620259	17.9878195
15	36.602140042459581 -121.8740217879334	36.602140074719522 -121.8740217778964	0.0036948523728406093	1.61716852
16	36.602141838846023 -121.8740210326680	36.602141791885543 -121.8740210700294	0.0061917889891614196	2.67141797

Table 14. Phase II Test Results: Distance between estimated interception point and calculated location

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

A. SUMMARY AND CONCLUSIONS

We began with an introduction to WSNs and the evidence supporting the vast potential this technology has to benefit a wide range of surveillance applications. We then discussed UAV—WSN integration and the motivations driving research toward this end.

To understand the benefits of this endeavor, one must first become familiar with the current state of WSN technology, the progress made in WSN contact detection and identification mechanisms, and the UAV equipment available for use in this capacity. The Background chapter covered these topics in great detail. Aside from a general understanding, it is important for the reader to appreciate the power management and security issues relating to WSNs. The survivability of this technology is an important factor to consider during operational planning, especially since these systems will be deployed in hostile environments in many military and law enforcement applications.

The OTAv1 and TRSSv4 were presented to familiarize the reader with the prior thesis work completed in WSN contact detection and classification. OTAv1 is responsible for parsing and evaluating the WSN output stream. This application detects WSN contacts by comparing WSN sensor readings to predefined thresholds, while TRSSv3 directs cameras within the WSN to take pictures of each contact. While these pictures will substantially aid the user's contact classification efforts, some circumstances require further investigation of the contact. UAVs provide an efficient and inexpensive solution to this problem.

We approached UAV technology with the scope of this thesis in mind. Both the MMALV and Procerus Unicorn utilize the Kestrel Autopilot System, for which PCPTAv1 was designed. MMALV is a state of the art micro UAV that combines the award winning flight design of the UF MAV with an efficient terrestrial locomotion solution developed at Case Western Reserve University. This versatile platform could serve in an investigative role, working in conjunction with a WSN to classify contacts, or

could actually function as a node within an adaptable WSN. The Procerus Unicorn was used during the development of PCPTAv1 as a test bed to evaluate the precision of its output.

A discussion of the Kestrel Autopilot System closed out the Background chapter. This system includes all the hardware and software required to fly a UAV autonomously. Kestrel Autopilot is the "brain" of the UAV, controlling its various functions according to user input and the autonomous control offered within VC. The VCDI allows user applications to send command packets to the Kestrel Autopilot through VC. Users interface with the Kestrel Autopilot System at a ground station, which transmits and receives packets from the Kestrel Autopilot through a Procerus Commbox.

In Chapter III, PCPTAv1 was introduced to the reader. PCPTAv1 represents the main contribution of this thesis, and a large step towards the implementation of a fully integrated UAV—WSN system. Upon activation by a WSN contact report, PCPTAv1 can autonomously send a UAV to either the instigated WSN cluster or to an interception point along the contact's estimated path. All calculated paths are optimized in two dimensions.

The thesis continued in Chapter IV with a detailed report and analysis of the testing conducted to evaluate PCPTAv1 accuracy and program correctness. Phase I testing, which assessed PCPTAv1's Kestrel command packet crafting methods and flight plan optimization scheme was a resounding success. In all forty-four Phase I test cases, PCPTAv1 output was packetized according to the Kestrel Autopilot System specification and was optimized in two dimensions. Phase II testing evaluated the accuracy of PCPTAv1's contact interception point calculations. While this phase resulted in greater error than what was theoretically to be expected, the recorded levels of inaccuracy would not impede the UAV's ability to acquire a visual on the contact during an interception scenario.

The vantage point UAVs offer as well as the relative ease with which it can be modified to attain the most telling imagery of a WSN contact, make the UAV an effective tool for WSN contact classification. This thesis has successfully designed and tested an application that takes advantage of that fact.

B. RECOMMENDATIONS FOR FUTURE WORK

Much work remains to be done before it is possible to deploy a fully integrated autonomous UAV—WSN system. One need not look further than the PCPTAv1 assumptions detailed in Chapter III to find worthwhile ventures in this space. For PCPTAv1 to acquire the input it needs to guide autonomous UAV surveillance operations, OTAv1 must be expanded upon to function within any WSN node layout. GPS data is also a necessity that OTAv1 does not currently provide due to the WSN hardware for which it was developed.

PCPTAv1 was the first attempt at an application providing autonomous UAV contact classification support for WSNs in an ongoing project at the Naval Postgraduate School. Development will continue, as this application is by no measure complete. Future development must focus first on the manner in which contact interception is calculated. Libraries capable of solving a system of nonlinear equations should be added to PCPTAv1, and interception location should be calculated in the manner discussed in Chapter IV and Appendix B. An accurate method for conversion from relative position to the geographic coordinate system would also be needed for the application to benefit from the precision of these calculations.

The flight plans produced in PCPTAv1 do not account for the topology of the area of operation. Optimized paths are constructed in two dimensions, but PCPTAv1 does not currently address the third. An algorithm that attempts to minimize terrain height gradients as well as the length of the path traveled to a destination remains to be implemented. As do the mechanisms required for collision detection and avoidance.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. USER DEFINED VIRTUAL COCKPIT DEVELOPMENT INTERFACE COMMAND PACKET STRUCTURE GUIDE

Byte Index	Type	Name	Description	Value
N/A	N/A	Header	Header; added by VC before packet is sent to the Kestrel Autopilot (not part of user defined packets)	0xff
0	32-Bit Integer	VCDI Packet Type	Indicates either Passthrough packet (10) or Packet Forwarding Setup packet (20)	10
4	32-Bit Integer	Packet Size	Number of bytes from byte 8 to the end of the packet	LAST BYTE minus 8
8	16-Bit Integer	Destination	Will hold the destination address of a Kestrel Autopilot	1032 (default)
10	UCHAR	Kestrel Packet Type	Defines the type of the Kestrel Packet; Command Packets (50), Command Edit Packets (53), etc.)	50
11	UCHAR	Command Type	Either a Goto, Loiter or Jump Command	Varies
12	UCHAR	Command Number	Number describing the position of this command in the flight plan	Varies

13	UCHAR	Total Commands	Total number of commands in the current flight plan	Varies
14	VARIOUS	Command Specific Bytes	Refer to Tables 4, 5 and 6	Varies
LAST BYTE	UCHAR	Payload	For future use, but must be included in the packet as a place holder	0
N/A	N/A	Packet ID	Distinguishes one packet from another; added by VC before packet is sent to the Kestrel Autopilot (not part of user defined packets)	Varies
N/A	N/A	XOR Check	Check to maintain packet integrity; added by VC before packet is sent to the Kestrel Autopilot (not part of user defined packets)	Varies
N/A	N/A	Footer	Footer; added by VC before packet is sent to the Kestrel Autopilot (not part of user defined packets)	0xfe

APPENDIX B. MAPLE 10 PHASE II TEST CASE CALCULATIONS

1.

> $A := 353.55339 + 20 \cdot t \cdot .707106 - 15 \cdot t \cdot \sin(h);$

$$353.55339 + 14.142120\,t - 15\,t\,\sin(h)$$

$B := 353.55339 + 20 \cdot t \cdot .707106 - 15 \cdot t \cdot \cos(h);$

$$353.55339 + 14.142120\,t - 15\,t\,\cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = -14.28572328,\ h = -2.356194490],\ [t = -100.0004417,\ h = 0.7853981634]]$$

2.

> $A := 353.55339 + 5 \cdot t \cdot .9205 - 15 \cdot t \cdot \sin(h);$

$$353.55339 + 4.6025\,t - 15\,t\,\sin(h)$$

$B := 353.55339 + 5 \cdot t \cdot .39073 - 15 \cdot t \cdot \cos(h);$

$$353.55339 + 1.95365\,t - 15\,t\,\cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 48.79618653,\ h = 0.9105929386],\ [t = -25.61672440,\ h = -2.481389265]]$$

3.

> $A := 353.55339 - 25 \cdot t \cdot .342020 - 15 \cdot t \cdot \sin(h);$

$$353.55339 - 8.550500\,t - 15\,t\,\sin(h)$$

$B := 353.55339 - 25 \cdot t \cdot .939693 - 15 \cdot t \cdot \cos(h);$

$$353.55339 - 23.492325\,t - 15\,t\,\cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 15.01263570,\ h = 1.566925476],\ [t = 41.63155692,\ h = -3.137721802]]$$

4.

> $A := 353.55339 - 10 \cdot t \cdot .173648 - 15 \cdot t \cdot \sin(h);$

$$353.55339 - 1.736480\,t - 15\,t\,\sin(h)$$

$B := 353.55339 - 10 \cdot t \cdot .984808 - 15 \cdot t \cdot \cos(h);$

$$353.55339 - 9.848080\,t - 15\,t\,\cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 22.67412202,\ h = 1.177773695],\ [t = -88.20631164,\ h = -2.748570022]]$$

5.

> $A := -353.55339 - 16 \cdot t \cdot .939692 - 15 \cdot t \cdot \sin(h);$

$$-353.55339 - 15.035072\ t - 15\ t\ \sin(h)$$

$B := 353.55339 + 16 \cdot t \cdot .342020 - 15 \cdot t \cdot cos(h);$

$$353.55339 + 5.472320\ t - 15\ t\ \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = -17.92734897,\ h = 2.823847404],\ [t = -449.8490180,\ h = -1.253051077]]$$

6.

> $A := -353.55339 - 14 \cdot t \cdot .342020 - 15 \cdot t \cdot \sin(h);$

$$-353.55339 - 4.788280\ t - 15\ t\ \sin(h)$$

$B := 353.55339 + 14 \cdot t \cdot .939693 - 15 \cdot t \cdot cos(h);$

$$353.55339 + 13.155702\ t - 15\ t\ \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 456.4175971,\ h = -.3799353884],\ [t = -18.88780252,\ h = 1.950731715]]$$

7.

> $A := -353.55339 + 30 \cdot t \cdot .573576 - 15 \cdot t \cdot \sin(h);$

$$-353.55339 + 17.207280\ t - 15\ t\ \sin(h)$$

$B := 353.55339 - 30 \cdot t \cdot .819152 - 15 \cdot t \cdot cos(h);$

$$353.55339 - 24.574560\ t - 15\ t\ \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 11.46511546,\ h = -1.140085211],\ [t = 32.30413628,\ h = 2.710881538]]$$

8.

> $A := -353.55339 + 2 \cdot t \cdot .999848 - 15 \cdot t \cdot \sin(h);$

$$-353.55339 + 1.999696\ t - 15\ t\ \sin(h)$$

$B := 353.55339 + 2 \cdot t \cdot .017452 - 15 \cdot t \cdot cos(h);$

$$353.55339 + 0.034904\,t - 15\,t\cos(h)$$

```
solve([A = 0, B = 0], [t, h]);
```

$$[[t = 30.63694499,\ h = -.6893385368],\ [t = -36.92345069,\ h = 2.260134864]]$$

9.

```
> A := -353.55339 - 100·t·.939693 - 20·t·sin(h);
```

$$-353.55339 - 93.969300\,t - 20\,t\sin(h)$$

```
B := -353.55339 - 100·t·.342020 - 20·t·cos(h);
```

$$-353.55339 - 34.202000\,t - 20\,t\cos(h)$$

```
solve([A = 0, B = 0], [t, h]);
```

$$[[t = -4.720350896 + 1.939055782\ I,\ h = -0.7853981634 + 1.379921070\ I],\ [t = -4.720350896 - 1.939055782\ I,\ h = -0.7853981634 - 1.379921070\ I]]$$

10.

```
> A := -353.55339 - 19·t·.087156 - 20·t·sin(h);
```

$$-353.55339 - 1.655964\,t - 20\,t\sin(h)$$

```
B := -353.55339 - 19·t·.996195 - 20·t·cos(h);
```

$$-353.55339 - 18.927705\,t - 20\,t\cos(h)$$

```
solve([A = 0, B = 0], [t, h]);
```

$$[[t = 389.6547793,\ h = -3.013073433],\ [t = -16.45121549,\ h = 1.442277107]]$$

11.

```
> A := -353.55339 + 50·t·.087156 - 20·t·sin(h);
```

$$-353.55339 + 4.357800\,t - 20\,t\sin(h)$$

```
B := -353.55339 + 50·t·.996195 - 20·t·cos(h);
```

$$-353.55339 + 49.809750\,t - 20\,t\cos(h)$$

```
solve([A = 0, B = 0], [t, h]);
```

$$[[t = 9.119574378 + 5.990065960\ I,\ h = -0.7853981634 + 1.052527834\ I],\ [t = 9.119574378 - 5.990065960\ I,\ h = -0.7853981634 - 1.052527834\ I]]$$

12.

> $A := -353.55339 - 15 \cdot t \cdot .707107 - 20 \cdot t \cdot \sin(h);$

$$-353.55339 - 10.606605\, t - 20\, t \sin(h)$$

$B := -353.55339 + 15 \cdot t \cdot .707107 - 20 \cdot t \cdot \cos(h);$

$$-353.55339 + 10.606605\, t - 20\, t \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 37.79646228, h = -1.508132060], [t = -37.79646228, h = -0.06266426647]]$$

13.

> $A := 353.55339 + 32 \cdot t \cdot 0 - 30 \cdot t \cdot \sin(h);$

$$353.55339 - 30\, t \sin(h)$$

$B := -353.55339 - 32 \cdot t \cdot 1 - 30 \cdot t \cdot \cos(h);$

$$-353.55339 - 32\, t - 30\, t \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = -11.81331205, h = -1.639905048], [t = -170.6658570, h = -3.072483932]]$$

14.

> $A := 353.55339 + 25 \cdot t \cdot .996195 - 30 \cdot t \cdot \sin(h);$

$$353.55339 + 24.904875\, t - 30\, t \sin(h)$$

$B := -353.55339 - 25 \cdot t \cdot .087156 - 30 \cdot t \cdot \cos(h);$

$$-353.55339 - 2.178900\, t - 30\, t \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 80.88048722, h = 1.790909649], [t = -11.23994517, h = -.2201133224]]$$

15.

> $A := 353.55339 - 35 \cdot t \cdot .984808 - 30 \cdot t \cdot \sin(h);$

$$353.55339 - 34.468280\, t - 30\, t \sin(h)$$

$B := -353.55339 + 35 \cdot t \cdot .173648 - 30 \cdot t \cdot \cos(h);$

$$-353.55339 + 6.077680\, t - 30\, t \cos(h)$$

$solve([A = 0, B = 0], [t, h]);$

$$[[t = 9.810937985, h = 3.089289635], [t = 78.40529991, h = -1.518493308]]$$

16.

```
> A := 353.55339 + 13·t·.087156 − 30·t·sin(h);
```

$$353.55339 + 1.133028\,t - 30\,t\,\sin(h)$$

```
B := −353.55339 + 13·t·.996195 − 30·t·cos(h);
```

$$-353.55339 + 12.950535\,t - 30\,t\,\cos(h)$$

```
solve([A = 0, B = 0], [t, h]);
```

$$[[t = 13.64065931,\ h = 2.017821518],\ [t = -25.07190497,\ h = -.4470251912]]$$

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. PCPTAV1 CODE

```
////////////////////////////////////////////////////////////
//Name: DevDemo.cpp
//
//Author: Procerus Technologies (http://www.procerusuav.com/)
//
//Description: Initializes the Windows application and defines
//             the class behaviors for the application.
////////////////////////////////////////////////////////////
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "DevDemo.h"
#include "DevDemoDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CDevDemoApp
BEGIN_MESSAGE_MAP(CDevDemoApp, CWinApp)
      ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

// CDevDemoApp construction
CDevDemoApp::CDevDemoApp()
{

      // Place all significant initialization in InitInstance
}

// The one and only CDevDemoApp object
CDevDemoApp theApp;

// CDevDemoApp initialization
BOOL CDevDemoApp::InitInstance()
{
      // InitCommonControls() is required on Windows XP if an
      // application; manifest specifies use of ComCtl32.dll version 6
      // or later to enable visual styles.  Otherwise, any window
      // creation will fail.
      InitCommonControls();

      CWinApp::InitInstance();

      if (!AfxSocketInit())
      {
            AfxMessageBox(IDP_SOCKETS_INIT_FAILED);
            return FALSE;
      }
```

```
	AfxEnableControlContainer();

	// Standard initialization
	// If you are not using these features and wish to reduce the
	// size of your final executable, you should remove from the
	// following the specific initialization routines you do not need
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));

	CDevDemoDlg dlg;
	m_pMainWnd = &dlg;
	INT_PTR nResponse = dlg.DoModal();
	if (nResponse == IDOK)
	{

		//  dialog is dismissed with OK
	}
	else if (nResponse == IDCANCEL)
	{
		// dialog is dismissed with Cancel
	}

	// Since the dialog has been closed, return FALSE so that we exit
	// the application, rather than start the application's message
	// pump.
	return FALSE;

	}
```

//

```
//Name: DevDemo.cpp
//
//Author: Procerus Technologies (http://www.procerusuav.com/)
//
//Description: Main Windows application header file
////////////////////////////////////////////////////////////
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////

#include "resource.h"         // main symbols

#pragma once

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

// CDevDemoApp:
// See DevDemo.cpp for the implementation of this class
class CDevDemoApp : public CWinApp
{
      public:
            CDevDemoApp();

      // Overrides
      public:
            virtual BOOL InitInstance();

            // Implementation
            DECLARE_MESSAGE_MAP()
};


      extern CDevDemoApp theApp;









////////////////////////////////////////////////////////////
```

```
//Name: DevDemoDlg.cpp
//
//Authors: Procerus Technologies (www.procerusuav.com) and
//         Stephen Schall
//
//Description: Implementation file for VC packet crafting,
//             telemetry data acquisition, and button handlers.
//             OnBnClickedPumpData() initiates the UAV path-planning
//             operation.
////////////////////////////////////////////////////////////////
// Applicable to all code authored by Procerus Technologies:
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////////

#include <Windows.h>
#include "stdafx.h"
#include "DevDemo.h"
#include "DevDemoDlg.h"
#include <string>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

using namespace std;


// Author: Procerus Technologies
// Description: CAboutDlg dialog used for App About.
class CAboutDlg : public CDialog
{
      public:
      CAboutDlg();

      // Dialog Data
      enum { IDD = IDD_ABOUTBOX };

      protected:
      // DDX/DDV support
      virtual void DoDataExchange(CDataExchange* pDX);

      // Implementation
      protected:
      DECLARE_MESSAGE_MAP()
};


// Author: Procerus Technologies
// Description: CAboutDlg constructor.
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}
```

```
// Author: Procerus Technologies
// Description: CAboutDlg dialog data exchange.
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}


BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()


// Author: Procerus Technologies
// Description: CDevDemoDlg dialog.
CDevDemoDlg::CDevDemoDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CDevDemoDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
	m_hSmallIcon = (HICON) LoadImage(AfxGetInstanceHandle(),
		(LPCTSTR)IDR_MAINFRAME,IMAGE_ICON,16,16,0);
	m_VCConnector = NULL;

	m_UAVAddress = 1032; //default uav address
}


// Author: Procerus Technologies
// Description: CDevDemoDlg dialog data exchange.
void CDevDemoDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
}

//Dialog message mapping
BEGIN_MESSAGE_MAP(CDevDemoDlg, CDialog)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_BN_CLICKED(IDCANCEL, OnBnClickedCancel)
	ON_BN_CLICKED(IDOK, OnBnClickedOk)
	ON_COMMAND(IDMF_EXIT, OnExit)
	ON_COMMAND(IDM_ABOUTBOX, OnAboutbox)
	ON_COMMAND(IDMF_CONNECT, OnConnect)
	ON_BN_CLICKED(IDC_CMD_ZERO_PRESS, OnBnClickedCmdZeroPress)
	ON_BN_CLICKED(IDC_ALL_PACKETS_RADIO, OnAllPackets)
	ON_BN_CLICKED(IDC_ACK_RADIO, OnAcksOnly)
	ON_BN_CLICKED(IDC_STANDARD_ACK_RADIO, OnAcksStd)
	ON_MESSAGE(WM_DATA_MSG,OnVCMsg)
	ON_BN_CLICKED(IDC_BUTTON1, OnBnClickedPumpData)
	ON_BN_CLICKED(IDC_ADDRESS_BUTTON,
	&CDevDemoDlg::OnBnClickedAddressButton)
	END_MESSAGE_MAP()


// Author: Procerus Technologies, modified by Stephen Schall
// Description: CDevDemoDlg message handlers.
BOOL CDevDemoDlg::OnInitDialog()
```

```
{
	CDialog::OnInitDialog();

	//Initialize windows XP themes
	InitCommonControls();

	// Add "About..." menu item to system menu.
	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
			strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this
	// automatically when the application's main window is not a
	// dialog
	SetIcon(m_hIcon, TRUE);		// Set big icon
	SetIcon(m_hSmallIcon, FALSE); // Set small icon

	//Now lets try to create the connection socket to the VC
	m_VCConnector = new CSocketConnector("127.0.0.1",GetSafeHwnd());

	//Check to see if we are connected
	if(m_VCConnector->IsConnected() == FALSE)
	{
		delete m_VCConnector;
		m_VCConnector = NULL;
		SetWindowText("PCPTAv1 - Not Connected To Virtual
		Cockpit");
	}
	else
	{
		SetWindowText("PCPTAv1 - Connected To Virtual Cockpit");
		OnAllPackets();   //Default to VC forwarding all packets
		((CButton*)GetDlgItem(IDC_ALL_PACKETS_RADIO))->SetCheck(1);

	}

	//Set default Multiplier
	GetDlgItem(IDC_MULTIPLIER)->SetWindowText("50.0");

	//Set default UAV Max Speed
	GetDlgItem(IDC_MAX_SPEED)->SetWindowText("15.0");

	//Set default UAV Range
	GetDlgItem(IDC_RANGE)->SetWindowText("4.0");
```

```
	//Set default UAV Address for UAV-WSN System
	GetDlgItem(IDC_ADDRESS)->SetWindowText("1032");
	OnBnClickedAddressButton();

	return TRUE;
}


// Author: Procerus Technologies, modified by Stephen Schall
// Description: Handles system commands.
void CDevDemoDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID & 0xFFF0) == IDM_ABOUTBOX)
	{
		CAboutDlg dlgAbout;
		dlgAbout.DoModal();
	}
	else if (nID == SC_CLOSE)
	{
		DestroyWindow();
	}
	else
	{
		CDialog::OnSysCommand(nID, lParam);
	}
}


// Author: Procerus Technologies
// Description: If you add a minimize button to your dialog, you will
// need the code below to draw the icon.  For MFC applications using
// the document/view model, this is automatically done for you by the
framework.
void CDevDemoDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND,
		reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
```

```
	}
}


// Author: Procerus Technologies
// Description: The system calls this function to obtain the cursor to
// display while the user drags the minimized window.
HCURSOR CDevDemoDlg::OnQueryDragIcon()
{
	return static_cast<HCURSOR>(m_hIcon);
}


// Author: Procerus Technologies
// Description: Cancel button handler.
void CDevDemoDlg::OnBnClickedCancel()
{
	;
}


// Author: Procerus Technologies
// Description: Ok button handler.
void CDevDemoDlg::OnBnClickedOk()
{
	;
}


// Author: Procerus Technologies
// Description: Exit handler.
void CDevDemoDlg::OnExit()
{
	//Close dialog
	OnOK();
}


// Author: Procerus Technologies
// Description: Aboutbox handler.
void CDevDemoDlg::OnAboutbox()
{
	CAboutDlg dlgAbout;
	dlgAbout.DoModal();
}


// Author: Procerus Technologies
// Description: Establishes connection to the Virtual Cockpit
// Development Interface.
void CDevDemoDlg::OnConnect()
{
	if(m_VCConnector != NULL && m_VCConnector->IsConnected() == TRUE)
	{
		MessageBox("Already connect to the Virtual Cockpit");
		return;
	}
```

```
	if(m_VCConnector != NULL)
	{
		delete m_VCConnector;
	}

	m_VCConnector = new CSocketConnector("127.0.0.1",GetSafeHwnd());

	if(m_VCConnector->IsConnected() == FALSE)
	{
		MessageBox("Could not connect to the Virtual Cockpit");
		delete m_VCConnector;
		m_VCConnector = NULL;
		return;
	}

	//If we made it here we are connected
	SetWindowText("Development Demo - Connected To Virtual Cockpit");
	OnAcksOnly();	//Default to VC forwarding acks only
	((CButton*)GetDlgItem(IDC_ACK_RADIO))->SetCheck(1);
}


// Author: Procerus Technologies
// Description: Zero Pressure button handler.
void CDevDemoDlg::OnBnClickedCmdZeroPress()
{
	//Construct a zero pressure packet
	sVCPacket ZeroPressPkt;		//Interface packet
	sPassThrough PassPkt;		//Passthrough packet structure
	ZeroPressPkt.VCPacketType = VC_PASSTHROUGH;

	//Now make the kestrel packet according to the Kestrel
	//Communications Guide
	PassPkt.DestAddr = m_UAVAddress;
	PassPkt.PassData[0] = 34;	//Recalibrate packet type id
	PassPkt.PassData[1] = 0;	//Recalibrate pressure sensors

	//Total size of this packet data is 2 bytes + 1 short = 4 bytes
	ZeroPressPkt.DataSize = 4;

	//Copy the data over....don't exceed 1024
	memcpy(ZeroPressPkt.PktData, &PassPkt, ZeroPressPkt.DataSize);

	m_VCConnector->SendData(&ZeroPressPkt);

	//Also clear the ACK text edit window
	GetDlgItem(IDC_PRESSURE_ACK)->SetWindowText("");
}


// Author: Procerus Technologies
// Description: "All Packets" radio button handler.
void CDevDemoDlg::OnAllPackets()
{
	//Make sure we are connected
	if(m_VCConnector == NULL) return;
```

```
	//Send a forward setup packet
	sVCPacket ForwardSetup;		//Interface packet
	ForwardSetup.VCPacketType = VC_FRWD_PKT_SETUP;

	//First enable the acks packet forwarding
	ForwardSetup.PktData[0] = 0;  //All Packet Type
	ForwardSetup.PktData[1] = 1;  //Turn it on

	ForwardSetup.DataSize = 2;    //Always 2 for forward setup

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC
}


// Author: Procerus Technologies
// Description: "Acknowledgments Only" radio button handler.
void CDevDemoDlg::OnAcksOnly()
{
	//Make sure we are connected
	if(m_VCConnector == NULL) return;

	//Send a forward setup packet
	sVCPacket ForwardSetup;		//Interface packet
	ForwardSetup.VCPacketType = VC_FRWD_PKT_SETUP;

	//First enable the acks packet forwarding
	ForwardSetup.PktData[0] = 1;  //Ack Packet Type
	ForwardSetup.PktData[1] = 1;  //Turn it on

	ForwardSetup.DataSize = 2;    //Always 2 for forward setup

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC

	//Now disable the std telem packet just incase they were enabled
	ForwardSetup.PktData[0] = 249;	//Std Telem Packet Type
	ForwardSetup.PktData[1] = 0;	//Turn it off

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC

	//Shut off the all packets flag
	ForwardSetup.PktData[0] = 0;  //The all packets flag
	ForwardSetup.PktData[1] = 0;  //Turn it off

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC
}


// Author: Procerus Technologies
// Description: "Acks + Standard Telemetry" radio button handler.
void CDevDemoDlg::OnAcksStd()
{
	//Make sure we are connected
	if(m_VCConnector == NULL) return;

	//Send a forward setup packet
	sVCPacket ForwardSetup;		//Interface packet
```

```
	ForwardSetup.VCPacketType = VC_FRWD_PKT_SETUP;

	//First enable the acks packet forwarding
	ForwardSetup.PktData[0] = 1;  //Ack Packet Type
	ForwardSetup.PktData[1] = 1;  //Turn it on

	ForwardSetup.DataSize = 2;    //Always 2 for forward setup

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC

	//Now set std telem packets
	ForwardSetup.PktData[0] = 249; //Std Telem Packet Type
	ForwardSetup.PktData[1] = 1;   //Turn it off

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC

	//Shut off the all packets flag
	ForwardSetup.PktData[0] = 0;  //The all packets flag
	ForwardSetup.PktData[1] = 0;  //Turn it off

	m_VCConnector->SendData(&ForwardSetup);   //Send it to the VC
}


// Author: Procerus Technologies
// Description: Process incoming VC packets.
LRESULT CDevDemoDlg::OnVCMsg(WPARAM wParam, LPARAM lParam)
{
	sVCPacket NewPkt;

	//Process all the packets
	while(m_VCConnector->NumVCPackets())
	{
		NewPkt = m_VCConnector->GetNextVCPacket();
		ProccessVCPkt(&NewPkt);
		m_VCConnector->PopVCPacket();
	}

	return 0;
}


// Author: Procerus Technologies, modified by Stephen Schall
// Description: Process and parse incoming VC packets for telemetry
// and navigation data.
void CDevDemoDlg::ProccessVCPkt(sVCPacket *NewPkt)
{
	unsigned char PacketType;
	CString EditStr1, EditStr2;
	char CurTime[128];
	time_t now;
	time(&now);

	//Get the time
	strftime(CurTime, 128, "[%H:%M:%S]", localtime(&now));

	switch(NewPkt->VCPacketType)
```

```
	{
	case VC_FRWD_PKT_SETUP:		//Shouldn't happen ever
	break;
	//These are the raw packet values with special characters removed
	//and checksum passed from the VC...refer to Kestrel
	//Communications Guide
	case VC_PASSTHROUGH:

		//First lets get the packet type..should be at byte offset
1
		PacketType = NewPkt->PktData[1];

		//Update received packet editbox
		GetDlgItem(IDC_PACKETS_RECEVIED_EDIT)-
		>GetWindowText(EditStr1);
		EditStr2.Format("%s Received Packet from VC Type ---
		%u\r\n",CurTime,PacketType);
		EditStr2 += EditStr1;
		//clear edit box if it gets too big
		if(EditStr2.GetLength() > 16384) EditStr2.Empty();
		GetDlgItem(IDC_PACKETS_RECEVIED_EDIT)-
		>SetWindowText(EditStr2);

		//Check if this is an ack for the zero pressure
		if(PacketType == 1 && NewPkt->PktData[6] == 34)
		{
			//Received ack from our zero pressure button....lets
			//set the text
			GetDlgItem(IDC_PRESSURE_ACK)->SetWindowText("Received
			Ack");
			return;
		}

		//Check to see if this packet is standard telemetry and
		//destined for m_UAVAddress
		unsigned short PacketAddress;
		memcpy(&PacketAddress,&NewPkt->PktData[2],2);
		if(PacketAddress == m_UAVAddress && PacketType == 249)
		{
			//Grab out the important data

			//Roll
			short TempShort;
			unsigned short TempUShort;
			unsigned char TempUChar;
			float Roll;
			memcpy(&TempShort, &NewPkt->PktData[10],2);
			Roll = TempShort * 57.3f / 1000.0f; //Roll in degrees

			//Pitch
			float Pitch;
			memcpy(&TempShort, &NewPkt->PktData[12],2);
			Pitch = TempShort * 57.3f / 1000.0f;

			//Heading
			float Heading;
			memcpy(&TempShort, &NewPkt->PktData[14],2);
```

```
                        Heading = TempShort * 57.3f / 1000.0f;
                        curUAVInfo.BearingUAV=Heading;

                        //Alt
                        float Alt;
                        memcpy(&TempUShort, &NewPkt->PktData[6],2);
                        Alt = (TempUShort / 6.0f) - 1000.0f;
                        curUAVInfo.altitudeUAV=Alt;

                        //Airspeed
                        float Airspeed;
                        memcpy(&TempUChar, &NewPkt->PktData[8],1);
                        Airspeed = (TempUChar / 20.0f) - 10.0f;
                        curUAVInfo.SpeedUAV=Airspeed;

                        //Write the values out
                        CString ValStr;
                        ValStr.Format("%.1f", Roll);
                        GetDlgItem(IDC_STD_ROLL)->SetWindowText(ValStr);
                        ValStr.Format("%.1f", Pitch);
                        GetDlgItem(IDC_STD_PITCH)->SetWindowText(ValStr);
                        ValStr.Format("%.1f", Heading);
                        GetDlgItem(IDC_STD_HEADING)->SetWindowText(ValStr);
                        ValStr.Format("%.1f", Alt);
                        GetDlgItem(IDC_STD_ALT)->SetWindowText(ValStr);
                        ValStr.Format("%.1f", Airspeed);
                        GetDlgItem(IDC_STD_AIRSPEED)->SetWindowText(ValStr);
                }
                //If its a navigational packet, pull out the UAV's Lat and
                //Long
                if(PacketAddress == m_UAVAddress && PacketType == 248)
                {
                        //Get current latitude and longitude
                        float lat2de;
                        float long2de;
                        memcpy(&lat2de, &NewPkt->PktData[14],4);
                        memcpy(&long2de, &NewPkt->PktData[20],4);
                        //Write out lat and long
                        CString str;
                        str.Format("%.7f", lat2de);
                        GetDlgItem(IDC_LAT)->SetWindowText(str);
                        str.Format("%.7f", long2de);
                        GetDlgItem(IDC_LONG)->SetWindowText(str);

                        //Save lat and long values for pathplanner use
                        curUAVInfo.latUAV=lat2de;
                        curUAVInfo.lonUAV=long2de;
                }
        break;
        }
}


// Author: Stephen Schall
// Description: "IGNITION" button handler.  This function executes
// the path-planning operation, then packetizes the resultant flight
// plan and sends it to the UAV via the Virtual Cockpit Development
```

```
// Interface (VCDI).
void CDevDemoDlg::OnBnClickedPumpData()
{
	string conLine;
	ifstream contactFile;

	//Make sure we are connected to VC
	if(m_VCConnector == NULL)
	{
		AfxMessageBox("Not Connected to VC");
		return;
	}

	//Accept all packets to receive telemetry and navigation data
	//from the UAV
	OnAllPackets();

	AfxMessageBox("Searching for Sensor Network contacts");

	//Wait for a text file with contact data to appear
	while(!contactFile.is_open())
	{
		contactFile.open("contact.txt");
	}

	//Get the first line of the contact text file
	getline(contactFile, conLine);
	//Close the contact file
	contactFile.close();

	//Ask user whether they want the UAV to intercept the contact
	//or fly to the sensor network
	int answer;
	bool chase=false;
	answer = AfxMessageBox("Do you want to intercept the contact?  To
	intercept with the active UAV press yes.  To fly to and loiter
	over the instigated WSN cluster press no.  To do nothing press
	cancel.", MB_YESNOCANCEL, 0);
	if(answer==IDYES)
	{
		AfxMessageBox("Intercepting the contact!");
		chase=true;
	}
	else if(answer==IDNO)
	{
		AfxMessageBox("Flying to the SN!");
	}
	else
	{
		AfxMessageBox("Doing nothing!");
		return;
	}

	//prep the current UAV data for transfer to PathPlanner
	float UAVIn[6];
	UAVIn[0]=curUAVInfo.altitudeUAV;
	UAVIn[1]=curUAVInfo.BearingUAV;
```

```
UAVIn[2]=curUAVInfo.latUAV;
UAVIn[3]=curUAVInfo.lonUAV;
UAVIn[4]=curUAVInfo.SpeedUAV;

//get UAV's turn radius (meters)
CString mult1;
float mult2;
GetDlgItem(IDC_MULTIPLIER)->GetWindowText(mult1);
mult1.TrimRight("\t "); //trim off leading and ending spaces
mult1.TrimLeft("\t ");
mult2=atof(mult1);

//get the UAV's range (nautical miles)
CString maxRange;
float mRange;
GetDlgItem(IDC_RANGE)->GetWindowText(maxRange);
maxRange.TrimRight("\t "); //trim off leading and ending spaces
maxRange.TrimLeft("\t ");
mRange=atof(maxRange);

//get UAV's max speed (meters/second)
CString maxSpeed;
float mSpeed;
GetDlgItem(IDC_MAX_SPEED)->GetWindowText(maxSpeed);
maxSpeed.TrimRight("\t "); //trim off leading and ending spaces
maxSpeed.TrimLeft("\t ");
mSpeed=atof(maxSpeed);

//Creates a file with waypoints generated by the path-planner
//that will be uploaded to the UAV
string file;
PathPlanner Path(conLine, chase, UAVIn, mult2, mRange, mSpeed);
file=Path.planner();

//check to ensure the destination is reachable
if(file=="trash")
      return;

int numCmds; //number of command position in overall flight plan
int totCmds; //total number of commands in a flight plan
string line;

//open input stream from the file containing the flight plan
ifstream myfile (file.c_str());

//count number of lines (commands) in the flight plan
totCmds=countCommands(file);
numCmds=totCmds;

//get packet data from text file
if (myfile.is_open())
      {
            //discards space at beginning of path plan
                  getline (myfile,line);

            //process commands
            while (! myfile.eof() )
```

```
                {
                        //collect a line of data (one packet's worth of
                        // information)
                        getline (myfile,line);
                        //Send the packet to the UAV
                        sendPacket(line, numCmds, totCmds);
                }
                myfile.close();
        }
      //if file cant be opened
      else
            AfxMessageBox("unable to find file", MB_OK, 0);

}


// Author: Stephen Schall
// Description: Parses a line (command) into the proper types and
// variables, then sends the command to the VCDI
void CDevDemoDlg::sendPacket(string line, int& numCmds, int totCmds)
{
      char commandType1; //stores the command type
      stringstream ss(line);
      string buffer;

      //get first string in the line (contains the command type)
      ss >> buffer;
      commandType1=buffer[0];

      //if the line describes a GoTo command
      if(commandType1=='2')
      {
            //struct to hold packet data during collection
            GoToPacket p;

            //collect packet data into a struct (refer to Kestrel
            //Autopilot System guide for details)

            //destination address
            p.destinationAddress=m_UAVAddress;
            //packet type
            p.packetType=static_cast<unsigned char>(50);
            //command type
            p.commandType=static_cast<unsigned char>(2);
            //command number
            p.commandNumber=static_cast<unsigned char>(totCmds-
            numCmds);
            //total commands in flight plan
            p.totalNumber=static_cast<unsigned char>(totCmds);
            //altitude (meters * 10)
            ss >> buffer;
            p.altitude=static_cast<unsigned short>
            (atoi(buffer.c_str()));
            //airspeed (m/s * 2)
            ss >> buffer;
            p.airspeed=static_cast<unsigned char>
            (atoi(buffer.c_str()));
```

```
		//latitude (degrees)
		ss >> buffer;
		p.lat=atof(buffer.c_str());
		//longitude (degrees)
		ss >> buffer;
		p.lon=atof(buffer.c_str());
		//payload (place holder for future use)
		ss >> buffer;
		p.payload=static_cast<unsigned char>(atoi(buffer.c_str()));

		//Copy struct into packet
		sVCPacket VCPkt;
		VCPkt.VCPacketType=VC_PASSTHROUGH;
		VCPkt.DataSize=sizeof(GoToPacket);
		memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
		//patch holes in packet from memcpy
		for(int i=9; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		for(int i=9; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		for(int i=9; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		//send command packet to VCDI
		m_VCConnector->SendData(&VCPkt);
	}
	//if the line describes a Loiter command
	else if(commandType1=='4')
	{
		//struct to hold packet data during collection
		LoiterPacket p;

		//collect packet data into a struct (refer to Kestrel
		//Autopilot System guide for details)

		//destination address
		p.destinationAddress=m_UAVAddress;
		//packet type
		p.packetType=50;
		//command type
		p.commandType=4;
		//command number
		p.commandNumber=totCmds-numCmds;
		//total number of commands in flight plan
		p.totalNumber=totCmds;
		//altitude (meters * 10)
		ss >> buffer;
		p.altitude=static_cast<unsigned short>
		(atoi(buffer.c_str()));
		//airspeed (m/s * 2)
		ss >> buffer;
		p.airspeed=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//loiter time (seconds, 0 for indefinite)
		ss >> buffer;
		p.loiterTime=static_cast<unsigned short>
		(atoi(buffer.c_str()));
		//loiter radius (meters)
```

```
		ss >> buffer;
		p.loiterRadius=static_cast<unsigned short>
		//(atoi(buffer.c_str()));
		//latitude (degrees)
		ss >> buffer;
		p.lat=static_cast<float>(atof(buffer.c_str()));
		//longitude (degrees)
		ss >> buffer;
		p.lon=static_cast<float>(atof(buffer.c_str()));
		//payload (place holder for future use)
		ss >> buffer;
		p.payload=static_cast<unsigned char>(atoi(buffer.c_str()));

		//Copy struct to packet
		sVCPacket VCPkt;
		VCPkt.VCPacketType=VC_PASSTHROUGH;
		VCPkt.DataSize=sizeof(LoiterPacket);
		memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
		//patch holes in packet
		for(int i=9; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		for(int i=13; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		for(int i=13; i<35; i++)
			VCPkt.PktData[i]=VCPkt.PktData[i+1];
		//send command packet to VCDI
		m_VCConnector->SendData(&VCPkt);

	}
	//if the line describes a land command edit
	else if(commandType1=='6')
	{
		//struct to hold packet data during collection
		LandPacket p;

		//collect packet data into a struct (refer to Kestrel
		//Autopilot System guide for details)

		//UAV address
		p.destinationAddress=m_UAVAddress;
		//packet type
		p.packetType=53;
		//command type
		p.commandType=6;
		//command number
		p.commandNumber=234;
		//discarded (place holder)
		p.totalNumber=totCmds;
		//airspeed (m/s * 2)
		ss >> buffer;
		p.airspeed=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//flair height (m/s * 2)
		ss >> buffer;
		p.flairHeight=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//loiter radius (meters)
```

```
        ss >> buffer;
        p.loiterRadius=static_cast<unsigned short>
        (atoi(buffer.c_str()));
        //latitude (degrees)
        ss >> buffer;
        p.lat=static_cast<float>(atof(buffer.c_str()));
        //longitude (degrees)
        ss >> buffer;
        p.lon=static_cast<float>(atof(buffer.c_str()));

        //copy struct into packet
        sVCPacket VCPkt;
        VCPkt.VCPacketType=VC_PASSTHROUGH;
        VCPkt.DataSize=sizeof(LandPacket);
        memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
        //send edit command packet to VCDI
        m_VCConnector->SendData(&VCPkt);
}
//if the line describes a takeoff command edit
else if(commandType1=='7')
{
        //struct to hold packet data during collection
        TakeoffPacket p;

        //collect packet data into a struct (refer to Kestrel
        //Autopilot System guide for details)

        //UAV address
        p.destinationAddress=m_UAVAddress;
        //packet type
        p.packetType=50;
        //command type
        p.commandType=7;
        //command number
        p.commandNumber=233;
        //discarded (place holder)
        p.totalNumber=totCmds;
        //finish altitude (meters * 10)
        ss >> buffer;
        p.finAlt=static_cast<unsigned short>
        (atoi(buffer.c_str()));
        //climb out radius (meters)
        ss >> buffer;
        p.climbOutRadius=static_cast<unsigned short>
        (atoi(buffer.c_str()));
        //climb out airspeed (m/s * 2)
        ss >> buffer;
        p.climbOutAirspeed=static_cast<unsigned char>
        (atoi(buffer.c_str()));
        //latitude (degrees)
        ss >> buffer;
        p.lat=static_cast<float>(atof(buffer.c_str()));
        //longitude (degrees)
        ss >> buffer;
        p.lon=static_cast<float>(atof(buffer.c_str()));

        //copy struct into packet
```

```
        sVCPacket VCPkt;
        VCPkt.VCPacketType=VC_PASSTHROUGH;
        VCPkt.DataSize=19;
        memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
        //send edit command packet to VCDI
        m_VCConnector->SendData(&VCPkt);

}
//if the line describes a jump command
else if(commandType1=='8')
{
        //struct to hold packet data during collection
        JumpCommandPacket p;

        //collect packet data into a struct (refer to Kestrel
        //Autopilot System guide for details)

        //UAV address
        p.destinationAddress=m_UAVAddress;
        //packet type
        p.packetType=50;
        //command type
        p.commandType=8;
        //command number
        p.commandNumber=totCmds-numCmds;
        //total number of commands in flight plan
        p.totalNumber=totCmds;
        //command number to jump to
        ss >> buffer;
        p.jumpToCommandNumber=static_cast<unsigned char>
        (atoi(buffer.c_str()));

        //copy struct into packet
        sVCPacket VCPkt;
        VCPkt.VCPacketType=VC_PASSTHROUGH;
        VCPkt.DataSize=sizeof(JumpCommandPacket);
        memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
        //send jump command to VCDI
        m_VCConnector->SendData(&VCPkt);

}
//if the line describes a takeoff command edit
else if(commandType1=='1')
{
        //struct to hold packet data during collection
        LandApproachPacket p;

        //collect packet data into a struct (refer to Kestrel
        //Autopilot System guide for details)

        //UAV address
        p.destinationAddress=m_UAVAddress;
        //packet type
        p.packetType=50;
        //command type
        p.commandType=10;
        //command number to edit
```

```
		p.commandNumber=236;
		//total number of commands in the flight plan
		p.totalNumber=totCmds;
		//approach airspeed (m/s * 2)
		ss >> buffer;
		p.approachAirspeed=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//flair airspeed (m/s * 2)
		ss >> buffer;
		p.flairAirspeed=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//circle down radius (meters)
		ss >> buffer;
		p.circleDownRadius=static_cast<unsigned short>
		(atoi(buffer.c_str()));
		//circle descent rate (m/s * 10)
		ss >> buffer;
		p.circleDescentRate=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//approach altitude (meters * 10)
		ss >> buffer;
		p.approachAltitude=static_cast<unsigned short>
		(atoi(buffer.c_str()));
		//altitude UAV breaks out of approach loiter and follows
		// glide slope path
		ss >> buffer;
		p.breakoutAltitude=static_cast<unsigned short>
		(atoi(buffer.c_str()));
		//flair height (meters * 10)
		ss >> buffer;
		p.flairHeight=static_cast<unsigned char>
		(atoi(buffer.c_str()));
		//approach latitude (degrees)
		ss >> buffer;
		p.approachLat=static_cast<float>(atof(buffer.c_str()));
		//approach longitude (degrees)
		ss >> buffer;
		p.approachLon=static_cast<float>(atof(buffer.c_str()));
		//land latitude (degrees)
		ss >> buffer;
		p.landLat=static_cast<float>(atof(buffer.c_str()));
		//land longitude (degrees)
		ss >> buffer;
		p.landLon=static_cast<float>(atof(buffer.c_str()));

		//copy struct into packet
		sVCPacket VCPkt;
		VCPkt.VCPacketType=VC_PASSTHROUGH;
		VCPkt.DataSize=sizeof(LandApproachPacket);
		memcpy(VCPkt.PktData, &p, VCPkt.DataSize);
		//send command edit packet to VCDI
		m_VCConnector->SendData(&VCPkt);
}
else
{
		//Received a bad command
		AfxMessageBox("Command Invalid");
```

```
	}

	numCmds=numCmds-1;
	return;
}


// Author: Stephen Schall
// Description: Counts the number of commands in a flight plan
int CDevDemoDlg::countCommands(string fileName)
{
	string line1;
	ifstream myfile1 (fileName.c_str());
	int count=0;

	//open text file
	if (myfile1.is_open())
		{
			while (! myfile1.eof() )
			{
				//count each line of data (a packet's worth of
				// data)
				getline (myfile1,line1);
				count=count+1;
			}

			myfile1.close();
		}

	//subtract out the empty line at the top of the data file
	return (count-1);
}


// Author: Stephen Schall
// Description: Changes the active UAV address
void CDevDemoDlg::OnBnClickedAddressButton()
{
	//get active UAV address from user input
	CString address;
	GetDlgItem(IDC_ADDRESS)->GetWindowText(address);
	address.TrimRight("\t "); //trim off leading and ending spaces
	address.TrimLeft("\t ");
	m_UAVAddress = atof(address);

}
```

```
////////////////////////////////////////////////////////////
//Name: DevDemoDlg.h
//
//Authors: Procerus Technologies (www.procerusuav.com) and
//         Stephen Schall
//
//Description: Header file for VCDI packet crafting,
//             telemetry data acquisition, and button handlers.
//
////////////////////////////////////////////////////////////////
// Applicable to all code authored by Procerus Technologies:
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////////

#pragma once

#include "SocketConnector.h"
#include "afxwin.h"
#include "PathPlanner.h"
#include <string>
using namespace std;


// Author: Stephen Schall
// Description: Data structure to hold current UAV data.
struct UAV2
{
      //current UAV latitude
      float latUAV;
      //current UAV longitude
      float lonUAV;
      //current UAV bearing (degrees)
      float BearingUAV;
      //current UAV velocity (m/s)
      float SpeedUAV;
      //current UAV altitude (meters)
      float altitudeUAV;

};


// Author: Procerus Technologies, modified by Stephen Schall
// Description: CDevDemoDlg dialog.
class CDevDemoDlg : public CDialog
{
      // Construction
      public:
            //Standard constructor
            CDevDemoDlg(CWnd* pParent = NULL);

            //Dialog Data
            enum { IDD = IDD_DEVDEMO_DIALOG };

      protected:
```

```
		// DDX/DDV support
		virtual void DoDataExchange(CDataExchange* pDX);

	// Implementation
	protected:
		HICON m_hIcon;
		HICON m_hSmallIcon;
		//Address all packets are sent to and received from
		unsigned short          m_UAVAddress;
		//Pointer to socket connection class
		CSocketConnector  *m_VCConnector;

		//Process the packets
		void ProccessVCPkt(sVCPacket *NewPkt);

		// Generated message map functions
		virtual BOOL OnInitDialog();
		afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
		afx_msg void OnPaint();
		afx_msg HCURSOR OnQueryDragIcon();
		DECLARE_MESSAGE_MAP()
	public:
		afx_msg void OnBnClickedCancel();
		afx_msg void OnBnClickedOk();
		afx_msg void OnExit();
		afx_msg void OnAboutbox();
		afx_msg void OnConnect();
		afx_msg void OnBnClickedCmdZeroPress();
		afx_msg void OnAllPackets();
		afx_msg void OnAcksOnly();
		afx_msg void OnAcksStd();


		//Function that gets called when a TCP/IP message is
		//received from the VC
		afx_msg LRESULT OnVCMsg (WPARAM wParam, LPARAM lParam);

		afx_msg void OnChangeDestAddr();
		afx_msg void OnBnClickedPumpData();

	public:
		void sendPacket(string line, int& numCmds, int totCmds);

	public:
		int countCommands(string fileName);

		//Holds current UAV telemetry and navigational data
		UAV2 curUAVInfo;
	public:
		afx_msg void OnBnClickedAddressButton();
};


// Author: Stephen Schall
// Description: Data structure to hold VCDI GoTo command packet data.
struct GoToPacket
{
```

```
            //packet destination address
            unsigned short destinationAddress;
            //packet type (50 for commands)
            unsigned char packetType;
            //command type (2 for GoTo commands)
            unsigned char commandType;
            //command number in the flight plan
            unsigned char commandNumber;
            //total number of commands in the flight plan
            unsigned char totalNumber;
            //altitude (meters * 10)
            unsigned short altitude;
            //airspeed (m/s * 2)
            unsigned char airspeed;
            //latitude (degrees)
            float lat;
            //longitude (degrees)
            float lon;
            //required as a place holder for future use
            unsigned char payload;

};


// Author: Stephen Schall
// Description: Data structure to hold VCDI Loiter command packet data.
struct LoiterPacket
{
            //packet destination address
            unsigned short destinationAddress;
            //packet type (50 for commands)
            unsigned char packetType;
            //command type (4 for Loiter commands)
            unsigned char commandType;
            //number of the command in the flight plan
            unsigned char commandNumber;
            //total number of commands in the flight plan
            unsigned char totalNumber;
            //altitude (meters * 10)
            unsigned short altitude;
            //airspeed (m/s * 2)
            unsigned char airspeed;
            //amount of time to loiter (seconds)
            unsigned short loiterTime;
            //radius of loiter circle (meters)
            unsigned short loiterRadius;
            //latitude of loiter circle center (degrees)
            float lat;
            //longitude of loiter circle center (degrees)
            float lon;
            //included as a place holder for future use
            unsigned char payload;
};


// Author: Stephen Schall
// Description: Data structure to hold VCDI Land Legal Circle command
```

```
// edit packet data.
struct LandPacket
{
            //packet destination address
            unsigned short destinationAddress;
            //packet type (53 for command edit)
            unsigned char packetType;
            //command type (6 for Land Legal Circle commands)
            unsigned char commandType;
            //command number to edit (234 for Land Legal Circle
            //commands)
            unsigned char commandNumber;
            //discarded, as there is only one packet in an edit
            //transmission
            unsigned char totalNumber;
            //airspeed of uav when landing (m/s * 2)
            unsigned char airspeed;
            //flair height (m/s * 2)
            unsigned char flairHeight;
            //radius of landing cirle loiter (meters)
            unsigned short loiterRadius;
            //latitude of land location (degrees)
            float lat;
            //longitude of land location (degrees)
            float lon;

};


// Author: Stephen Schall
// Description: Data structure to hold VCDI Takeoff command edit
// packet data.
struct TakeoffPacket
{
            //packet destination address
            unsigned short destinationAddress;
            //packet type (53 for command edit)
            unsigned char packetType;
            //command type (7 for Takeoff commands)
            unsigned char commandType;
            //command number to edit (233 for Takeoff command)
            unsigned char commandNumber;
            //discarded, as there is only one packet in an edit
            //transmission
            unsigned char totalNumber;
            //final altitude after takeoff (meters * 10)
            unsigned short finAlt;
            //radius of climb out circle (meters)
            unsigned short climbOutRadius;
            //airspeed of climb (m/s * 2)
            unsigned char climbOutAirspeed;
            //takeoff waypoint latitude (degrees)
            float lat;
            //takeoff waypoint longitude (degrees)
            float lon;

};
```

```
// Author: Stephen Schall
// Description: Data structure to hold VCDI Jump command packets.
struct JumpCommandPacket
{
            //packet destination address
            unsigned short destinationAddress;
            //packet type (50 for commands)
            unsigned char packetType;
            //command type (8 for jump commands
            unsigned char commandType;
            //command number in flight plan
            unsigned char commandNumber;
            //total number of commands in flight plan
            unsigned char totalNumber;
            //command number to jump to in flight plan
            unsigned char jumpToCommandNumber;

};


// Author: Stephen Schall
// Description: Data structure to hold VCDI Land Approach command edit
// packet data.
struct LandApproachPacket
{
            //packet destination address
            unsigned short destinationAddress;
            //packet type (53 for command edit)
            unsigned char packetType;
            //command type (10 for Takeoff commands)
            unsigned char commandType;
            //command number to edit (236 for Takeoff command)
            unsigned char commandNumber;
            //discarded, as there is only one packet in an edit
            //transmission
            unsigned char totalNumber;
            //approach airspeed (m/s * 2)
            unsigned char approachAirspeed;
            //airspeed used for flair and glide slope (m/s * 2)
            unsigned char flairAirspeed;
            //radius of circle used to descend (meters)
            unsigned short circleDownRadius;
            //airspeed of descent circle (m/s * 10)
            unsigned char circleDescentRate;
            //UAV altitude to approach waypoint (meters * 10)
            unsigned short approachAltitude;
            //altitude UAV breaks out of descent circle and into glide
            //slope (meters * 10)
            unsigned short breakoutAltitude;
            //flair height (meters * 10)
            unsigned char flairHeight;
            //approach waypoint latitude (degrees)
            float approachLat;
            //approach waypoint longitude (degrees)
            float approachLon;
```

```
            //land waypoint latitude (degrees)
            float landLat;
            //land waypoint longitude (degrees)
            float landLon;
};
```

```
////////////////////////////////////////////////////////////
```

```
//Name: DevDemoDlg.cpp
//
//Authors: Stephen Schall
//
//Description: This file contains the PathPlanner class, which
//                    creates a flight plan to either guide a UAV to a
//                    specific latitude and longitude (in operational
//             support UAV-WSN System scenarios), or intercept
//             a WSN contact (in UAV-WSN System contact
//             interception scenarios).  The action taken is
//             determined by the user, and the flight plan
//             output can be viewed in "dat.txt."
////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "PathPlanner.h"


//Author: Stephen Schall
//Description: This constructor feeds UAV and user input into the
//PathPlanner class.
PathPlanner::PathPlanner(string con, bool chase1, float uavIn[], float
mult, float mRange, float mSpeed)
{
      //UAV turn radius, taken from GUI (meters)
      turnRadius=mult;
      //UAV current latitude (degrees)
      curLat=uavIn[2];
      //UAV current longitude (degrees)
      curLon=uavIn[3];
      //Initiate UAV destination latitude
      destLat=0;
      //Initialize UAV destination longitude
      destLon=0;
      //UAV current bearing (degrees)
      curBearing=uavIn[1];
      //Initialize UAV destination bearing
      destBearing=0;
      //Initialize variable to store distance to destination;
      //may be WSN or contact intercept location
      distance=0;
      //Boolean value dictating whether to intercept contact or go
      //to instigated WSN
      follow=chase1;
      //UAV current speed (meters/second * 2)
      uavSpeed=uavIn[4];
      //String containing contact data: bearing (degrees),
      //velocity (meters/second), latitude (degrees), and
      //longitude (degrees)
      contct=con;
      //UAV current altitude
      uav1.altitudeUAV=uavIn[0];
      //Distance from UAV to WSN
      distanceToSN=0;
      //Sets the precision of latitude and longitude calculations
      out.precision(10);
      //UAV max range (nautical miles)
```

```
	maxRange=mRange;
	//UAV max speed (meters/second)
	maxSpeed=mSpeed;
}


//Author: Stephen Schall
//Description: PathPlanner destructor.
PathPlanner::~PathPlanner(void)
{
	return;
}


//Author: Stephen Schall
//Description: This is the main function driving flight plan
//creation.  It parses WSN input, calculates the bearing to
//the user defined destination, and creates a near-optimal
//two dimensional path to the destination.  This flight plan
//is then saved in "dat.txt."
string PathPlanner::planner()
{
	//Create or open text file to store flight plan
	string file2="dat.txt";
	out.open(file2.c_str());

	//Get WSN input
	in=sensorNwInput();

	//Calculate distance from UAV to WSN
	distanceToSN=measureDistance(in.latNW, in.lonNW, curLat, curLon);

	//Calculate bearing to target location
	destBearing = getInterceptBearing();

	//Check if it is possible to reach destination
	if(destBearing==-1)
	{
		AfxMessageBox("The destination is out of range");
		return "trash";
	}

	//Construct waypoint path to destination
	int trash=pathDecide(0);

	//Close the output file holding the flight plan
	out.close();

	return "dat.txt";
}


//Author: Stephen Schall
//Description: Parses sensor network input from DevDemoDlg,
//which gets its data from "contact.txt," the hypothetical
//OTAv1 output file.
SNInput PathPlanner::sensorNwInput(void)
```

```
{
	//Create a string stream for input
	stringstream ss2(contct);
	string buff;
	SNInput input;

	//Get contact's bearing (degrees)
	ss2 >> buff;
	input.contactBearing=atof(buff.c_str());
	//Get contact's velocity (meters/second)
	ss2 >> buff;
	input.contactSpeed=atof(buff.c_str());
	//Get contact's latitude (degrees)
	ss2 >> buff;
	input.latNW=atof(buff.c_str());
	//Get contact's longitude (degrees)
	ss2 >> buff;
	//Get contact's latitude (degrees)
	input.lonNW=atof(buff.c_str());

	//Load contact data into PathPlanner variables
	tarBearing=input.contactBearing;
	tarSpeed=input.contactSpeed;
	tarLat=input.latNW;
	tarLon=input.lonNW;

	//return contact data
	return input;
}


//Author: Stephen Schall
//Description: Computes the bearing the UAV will use
//to reach its destination.
float PathPlanner::getInterceptBearing()
{
	//Holds UAV bearing
	float interceptBearing;
	//Variables used for calculation
	double lat1, lat2, lon1, lon2, d;
	//Get currrent UAV position
	lat1=curLat;
	lon1=curLon;
	//Get WSN location
	lat2=tarLat;
	lon2=tarLon;
	//Get distance from UAV to WSN
	d=distanceToSN;

	//If the user has chosen to intercept the contact with a UAV
	if(follow)
	{
		//Number of points along contact's estimated path to poll
		//to see if the UAV can get there before it.  Sample size
		//accounts for 2 hours of contact movement, which is plenty
		//long considering the contact cannot be expected to
		//maintain the same exact bearing for 2 hours in most cases
```

```
const int numSamples=static_cast<const int>
((tarSpeed*3600*2)/5);
//Create an array of latitudes and an array of
//coorresponding longitudes to store the contact's
//estimated path
double * latSample;
latSample= new (nothrow) double[numSamples];
double * lonSample;
lonSample= new (nothrow) double[numSamples];
//Start contact estimated path at site of detection (WSN
//location)
latSample[0]=lat2;
lonSample[0]=lon2;
//Stores whether or not the contact is possible to
//intercept the contact
bool interceptPossible=false;
//Polls the estimated contact path every 5 meters; must be
//converted from meters to nm for calculations
double interval=5.0*.000539956803;
//Get time in seconds it takes contact to arrive at each
//possible intercept point
double time= (interval/.000539956803)/tarSpeed;
//Total time past in seconds
double timePast=0;
//Used to account for the amount of time it takes the UAV
//to turn to the bearing of the potential interception
//point
float timeAddedForTurn=0;
double intervalAdder=0;
//Do a simple test to see if the UAV can ever catch the
//contact
interceptPossible=possible(lat1, lon1, lat2, lon2,
tarBearing);

//If it may be possible to intercept the contact
if(interceptPossible)
{
      //Poll ever 5 meters along the contact's estimated
      //path to see where the UAV can intercept it
      for(int i=0; i<numSamples; i++)
      {
            //Generate possible intercept locations
            intervalAdder=interval+intervalAdder;
            nextPoint(lat2, lon2, tarBearing,
            intervalAdder, latSample[i+1], lonSample[i+1]);
            //Test to see if UAV can make it to intercept
            //location in time to meet contact
            interceptPossible=test(timePast,
            latSample[i+1], lonSample[i+1]);

            //If an achievable intercept is found, return
            //the bearing to it and update the destination
            if(interceptPossible)
            {
                  //Update destination to initial intercept
                  //location
                  destLat=latSample[i+1];
```

```
                destLon=lonSample[i+1];
                //Measure distance to initial intercept
                //location
                distance=measureDistance(lat1, lon1,
                latSample[i+1], lonSample[i+1]);
                //Get the initial intercept bearing to
                //intercept location
                interceptBearing=getBearing(lat1, lon1,
                latSample[i+1], lonSample[i+1]);
                //Account for the time it would take the
                //UAV to turn to the bearing of the
                //potential interception point

                timeAddedForTurn=addTimeForTurn
                (interceptBearing);
                double distanceChange=
                timeAddedForTurn*tarSpeed;
                //Convert from meters to nautical miles
                distanceChange=
                distanceChange*.000539956803;
                //If correction must be made to account
                //for turn time
                if(distanceChange>0)
                {
                      //Generate new intercept location
                      nextPoint(latSample[i],
                      lonSample[i], tarBearing,
                      distanceChange, latSample[i+2],
                      lonSample[i+2]);

                      //Once extra time is accounted for,
                      //get correct values
                      destLat=latSample[i+2];
                      destLon=lonSample[i+2];
                      distance=measureDistance(lat1,
                      lon1, latSample[i+2],
                      lonSample[i+2]);
                      interceptBearing=getBearing(lat1,
                      lon1, latSample[i+2],
                      lonSample[i+2]);
                      //Ensure the contact can still be
                      //intercepted if turn required

                      nextPoint(lat1, lon1, tarBearing,
                      distanceChange, lat2, lon2);

                      interceptPossible=possible(lat1,
                      lon1, lat2, lon2,
                      interceptBearing);
                      if(!interceptPossible)
                            return -1;
                }
                return interceptBearing;
          }
          //Add time it took contact to get to next poll
          //point along its estimated path
          timePast=time + timePast;
```

```
                        }
                        //If the contact cannot be intercepted with the UAV
                        interceptBearing=-1;
                }
                //If the contact cannot be intercepted with the UAV
                else
                {
                        interceptBearing=-1;
                }
        }
        //If the user selected to send a UAV to the instigated WSN
        else
        {
                //Check to ensure the sensor network is reachable
                if(distanceToSN>=maxRange)
                {
                        interceptBearing=-1;
                        return interceptBearing;
                }

                //Get bearing to WSN
                interceptBearing=getBearing(lat1, lon1, lat2, lon2);

                //Solidify destination
                distance=distanceToSN;
                destLat=tarLat;
                destLon=tarLon;
        }
        return interceptBearing;
}


//Author: Stephen Schall
//Description: Returns distance in nm between two points.  Equation
//from (Williams, 2004), (http://williams.best.vwh.net/avform)
double PathPlanner::measureDistance(double lat1, double lon1, double
lat2, double lon2)
{
        //Variable to hold distance between the points
        double dist;

        //Convert latitudes and longitudes to radian distances
        lat1=lat1*3.14159265/180.0;
        lat2=lat2*3.14159265/180.0;
        lon1=lon1*3.14159265/180.0;
        lon2=lon2*3.14159265/180.0;

        //Haversine formula
        dist=2*asin(sqrt(pow((sin((lat1-lat2)/2.0)),2.0) +
        cos(lat1)*cos(lat2)*(pow((sin((lon1-lon2)/2.0)),2.0))));
        //Convert distance from radians to nm
        dist=((180.0*60.0)/3.14159265)*dist;

        return dist;
}
```

```
//Author: Stephen Schall
//Description: Creates a flight plan to the UAV destination.
Determines
//whether to turn the UAV or not based on its bearing and its
destination.
//Returns a 1 if a turn is required, and 0 if not
int PathPlanner::pathDecide(int type)
{

	//Holds the returned value of makeTurn()
	int trash;
	//Holds whether the destination is straight ahead or not
	bool straightAhed=straightAhead();


	//If UAV bearing is toward quadrant I
	if(curBearing>=0 && curBearing<90)
	{
		//Check if turn greater than 45 degrees is required
		if(destBearing<=curBearing+45 || destBearing>=(360-(45-
		curBearing)))
		{
			//Turn greater than 45 degrees not required;
			//If distance from current location to destination
			//location is less than twice the turning radius, go
			//forward before turning, unless destination is
			//straight ahead
			if((distance<=((2*turnRadius)/1852.0)) &&
			(!straightAhed))
			{
				//If not for interception calculation
				if(type==0)
				{
					//Set goto point in front, then turn
					//around after moving forward
					makeGotoBeforeTurn();
					trash=makeTurn(0);
				}
				//Turn required
				return 1;
			}
			//Turn less than 45 degrees; distance to destination
			//greater than two times the turn radius of the UAV
			else
			{
				//If not for interception calculation
				if(type==0)
				{
					//Make Goto point packet
					makeGotoFlyStraight();
				}
			}
		}
		//Destination bearing requires harder than 45 degree turn
		else
		{
```

```
                //If distance from current location to destination
                //location is less than twice the turning radius
                if(distance<=((2*turnRadius)/1852.0))
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                //Set goto point in front, then turn
                                //around after moving forward
                                makeGotoBeforeTurn();
                                trash=makeTurn(0);
                        }
                }
                else
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                trash=makeTurn(0);
                        }
                }
                //Turn required
                return 1;
        }
}
//If UAV bearing is toward quadrants III or IV
else if(curBearing>=90 && curBearing<270)
{
        //Check if turn greater than 45 degrees is required
        if(destBearing>=(curBearing-45) &&
        destBearing<=(curBearing+45))
        {
                //Turn greater than 45 degrees not required;
                //If distance from current location to destination
                //location is less than twice the turning radius, go
                //forward before turning, unless destination is
                //straight ahead
                if(distance<=((2*turnRadius)/1852.0) &&
                (!straightAhed))
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                //Set goto point in front, then turn
                                //around after moving forward
                                makeGotoBeforeTurn();
                                trash=makeTurn(0);
                        }
                        //Turn required
                        return 1;
                }
                //Turn less than 45 degrees; distance to destination
                //greater than two times the turn radius of the UAV
                else
                {
                        //If not for interception calculation
                        if(type==0)
```

```
                        {
                                //Make Goto point packet
                                makeGotoFlyStraight();
                        }
                }
        }
        //Destination bearing requires harder than 45 degree turn
        else
        {
                //If distance from current location to destination
                //location is less than twice the turning radius
                if(distance<=((2*turnRadius)/1852.0))
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                //Set goto point in front, then turn
                                //around after moving forward
                                makeGotoBeforeTurn();
                                trash=makeTurn(0);
                        }
                }
                else
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                trash=makeTurn(0);
                        }
                }
                //Turn required
                return 1;
        }
}
//If UAV bearing is toward quadrant II
else if(curBearing>=270 && curBearing<360)
{
        //Check if turn greater than 45 degrees is required
        if(destBearing>=(curBearing-45) || destBearing<=(45-(360-
        curBearing)))
        {
                //Turn greater than 45 degrees not required;
                //If distance from current location to destination
                //location is less than twice the turning radius, go
                //forward before turning, unless destination is
                //straight ahead
                if(distance<=((2*turnRadius)/1852.0) &&
                (!straightAhed))
                {
                        //If not for interception calculation
                        if(type==0)
                        {
                                //Set goto point in front, then turn
                                //around after moving forward
                                makeGotoBeforeTurn();
                                trash=makeTurn(0);
                        }
```

```
                        //Turn required
                        return 1;
                  }
                  //Turn less than 45 degrees; distance to destination
                  //greater
                  //than two times the turn radius of the UAV
                  else
                  {
                        //If not for interception calculation
                        if(type==0)
                        {
                              //Send Goto point packet
                              makeGotoFlyStraight();
                        }
                  }
            }
            //Destination bearing requires harder than 45 degree turn
            else
            {
                  //If distance from current location to destination
                  //location is less than twice the turning radius
                  if(distance<=((2*turnRadius)/1852.0))
                  {
                        //If not for interception calculation
                        if(type==0)
                        {
                              //Set goto point in front, then turn
                              //around after moving forward
                              makeGotoBeforeTurn();
                              trash=makeTurn(0);
                        }
                  }
                  else
                  {
                        //If not for interception calculation
                        if(type==0)
                        {
                              trash=makeTurn(0);
                        }
                  }
                  //Turn required
                  return 1;
            }
      }
      else
            AfxMessageBox("ERROR computing path");

      return 0;
}


//Author: Stephen Schall
//Description: Makes a Loiter packet that flies the UAV "forward."
//Where "forward" in this case means that the UAV does not need to turn
//harder than 45 degrees to get to its destination.  Once at its
//destination, the UAV will circle overhead indefinitely until ordered
//to do otherwise.
```

```
void PathPlanner::makeGotoFlyStraight(void)
{
      //Write Loiter command to flight plan
      out<<"\n4 1200 "<<maxSpeed*2<<" 0 "<<turnRadius<<" "<<destLat<<"
      "<<destLon<<" 0";
}


//Author: Stephen Schall
//Description: Sends UAV four times its turn radius ahead to be able to
//turn for a destination that was within two times its turn radius from
//its current position.  Equation from (Williams, 2004),
//(http://williams.best.vwh.net/avform).
void PathPlanner::makeGotoBeforeTurn()
{
      //Variable declarations
      float interceptBearing, tc;
      float lat1, lat2, lon1, lon2, d;
      lat1=curLat;
      lon1=curLon;
      tc=curBearing;
      //Distance to travel ahead in nm
      d=(4*turnRadius)/1852.0;

      //Convert variables to radians
      tc=(3.14159265/180.0)*tc;
      d=(3.14159265/(180.0*60.0405))*d;
      lat1=lat1*3.14159265/180.0;
      lon1=-lon1*3.14159265/180.0;

      //Produces a new latitude and longitude given an original
      //position, a distance traveled and a bearing
      lat2=asin(sin(lat1)*cos(d)+cos(lat1)*sin(d)*cos(tc));
      if(cos(lat2)==0)
        lon2=lon1;
      else
        lon2=fmod(lon1-asin(sin(tc)*sin(d)/cos(lat2))
        +3.14159265,2*3.14159265)-3.14159265;

      //Convert back to degrees
      lat2=((lat2*180.0)/3.14159265);
      lon2=-((lon2*180.0)/3.14159265);

      //Write waypoint to flight plan
      out<<"\n2 1200 "<<maxSpeed*2<<" "<<lat2<<" "<<lon2<<" 0";

      //Update UAV current position
      curLat=lat2;
      curLon=lon2;
}


//Author: Stephen Schall
//Description: Handles any situation in which the UAV must turn
//harder than 90 degrees.  Variable type used to distinguish between
//write operation and intercept calculation calls.
int PathPlanner::makeTurn(int type)
```

```
{
	//Direction of turn
	bool flyRightCircle;
	//Holds the eight points representing the UAV turn circle
	double la[8];
	double lo[8];
	//Start turn with current position
	la[0]=curLat;
	lo[0]=curLon;
	//Make a 45 degree turn at each of the eight points
	//around the circle
	int turnAngle=45;
	//Keeps track of true bearing
	float trackAngle=0;
	//Used to know which point is the breakoff tangent
	bool breakOff=false;
	//Number of turns till breakoff
	int turnCount=0;

	//Determine whether to turn right or left to get to destination
	if(curBearing>=0 && curBearing<=180)
	{
		if(destBearing < curBearing || destBearing >=
		(curBearing+180))
			flyRightCircle=false;
		else
			flyRightCircle=true;
	}
	else
	{
		if(destBearing > curBearing || destBearing <=
		((curBearing+180)-360))
			flyRightCircle=true;
		else
			flyRightCircle=false;
	}

	//If turning right
	if(flyRightCircle)
	{
		//Retrieve current UAV true bearing
		trackAngle=curBearing;
		//Construct right turn circle
		for(int i=0; i<7; i++)
		{
			//Get true bearing for next turn around the circle
			if(trackAngle+turnAngle<=360)
				trackAngle=trackAngle+turnAngle;
			else
				trackAngle=trackAngle+turnAngle-360;

			//Get next point in the turn circle, radius of circle
			//is UAV turn radius.  Distance of each leg is
			//dependent upon UAV turn radius as well.
			nextPoint(la[i], lo[i], trackAngle,
			(((turnRadius)/1852.0)*sin(22.5*(3.14159265/180.0))),
			la[i+1], lo[i+1]);
```

```
                //Determine if this is the breakOff spot
                breakOff=poll(la[i+1], lo[i+1], trackAngle,
                flyRightCircle);

                //Write turn waypoints until the breakoff spot is
                //found and written
                if(!breakOff)
                {
                      //Add to turn count because UAV is still
                      //turning
                      turnCount++;
                      //If not an intercept calculation call
                      if(type==0)
                      {
                            //Write a Goto command for 1/8 of the
                            //turn circle
                            out<<"\n2 1200 "<<maxSpeed*2<<"
                            "<<la[i+1]<<" "<<lo[i+1]<<" 0";

                            //Update current location
                            curLat=la[i+1];
                            curLon=lo[i+1];
                      }
                }
                //If turn breakOff location found
                else
                {
                      //Still need to make the last turn
                      turnCount++;
                      //If not an intercept calculation call
                      if(type==0)
                      {
                            //Write last point of the turn
                            out<<"\n2 1200 "<<maxSpeed*2<<"
                            "<<la[i+1]<<" "<<lo[i+1]<<" 0";

                            //Fly to a point in between breakoff
                            //point and destination
                            float ptLat, ptLon;
                            intermediatPt(la[i+1], lo[i+1], destLat,
                            destLon, ptLat, ptLon);
                            out<<"\n2 1200 "<<maxSpeed*2<<"
                            "<<ptLat<<" "<<ptLon<<" 0";

                            //Fly to destination
                            makeGotoFlyStraight();
                            return 0;
                      }
                      //Return the magnitude of the turn
                      return turnCount;
                }
          }
}
//If turning left
else
{
```

```
//Retrieve current UAV true bearing
trackAngle=curBearing;

//Construct left circle
for(int i=0; i<7; i++)
{

	//Get true bearing for next turn around the circle
	if(trackAngle-turnAngle >= 0)
		trackAngle=trackAngle-turnAngle;
	else
		trackAngle=(360+(trackAngle-turnAngle));

	//Get next point in the turn circle, radius of circle
	//is UAV turn radius. Distance of each leg is
	//dependent upon UAV turn radius as well (fourth
	//parameter)
	nextPoint(la[i], lo[i], trackAngle,
	(((turnRadius)/1852.0)*sin(22.5*(3.14159265/180.0))),
	la[i+1], lo[i+1]);

	//Determine if this is the breakOff spot
	breakOff=poll(la[i+1], lo[i+1], trackAngle,
	flyRightCircle);

	//Write turn waypoints until the breakoff spot is
	found and written
	if(!breakOff)
	{
		//Add to turn count because UAV is still
		//turning
		turnCount++;
		//If not an intercept calculation call
		if(type==0)
		{
			//Write a Goto command for 1/8 of the
			//turn circle
			out<<"\n2 1200 "<<maxSpeed*2<<"
			"<<la[i+1]<<" "<<lo[i+1]<<" 0";
			//Update current UAV position
			curLat=la[i+1];
			curLon=lo[i+1];
		}
	}
	//If turn breakOff location found
	else
	{
		//Account for last point of turn
		turnCount++;
		//If not an interception calculation call
		if(type==0)
		{
			//Write last point of turn
			out<<"\n2 1200 "<<maxSpeed*2<<"
			"<<la[i+1]<<" "<<lo[i+1]<<" 0";
```

```
                                    //Fly to a point in between breakoff
                                    //point and destination
                                    float ptLat, ptLon;
                                    intermediatPt(la[i+1], lo[i+1], destLat,
                                    destLon, ptLat, ptLon);
                                    out<<"\n2 1200 "<<maxSpeed*2<<"
                                    "<<ptLat<<" "<<ptLon<<" 0";

                                    //Fly to destination
                                    makeGotoFlyStraight();
                                    return 0;
                              }
                              //Used to count the number of turns required to
                              //face a bearing
                              return turnCount;
                        }
                  }
            }
}


//Author: Stephen Schall
//Description: Computes a new latitude and longitude, a distance "d"
//away from the inputted location, at a specified true bearing.
//Equation from (Williams, 2004), (http://williams.best.vwh.net/avform)
void PathPlanner::nextPoint(double lat1, double lon1, double bearing,
double d, double& lat2, double& lon2)
{
      //Transfer bearing for ease of use
      double b=bearing;

      //Convert variables to radians
      b=(3.14159265/180.0)*b;
      d=(3.14159265/(180.0*60.0405))*d;
      lat1=lat1*3.14159265/180.0;
      //Made negative because this equation treats West coordinates as
      //positive
      lon1=-lon1*3.14159265/180.0;

      //Compute new latitude and longitude
      lat2=asin(sin(lat1)*cos(d)+cos(lat1)*sin(d)*cos(b));
    if(cos(lat2)==0)
        lon2=lon1;
       else
        lon2=fmod(lon1-asin(sin(b)*sin(d)/cos(lat2))
        +3.14159265,2*3.14159265)-3.14159265;

      //Convert back to degrees
      lat2=((lat2*180)/3.14159265);
      lon2=-((lon2*180)/3.14159265);
}


//Author: Stephen Schall
//Description: Decides whether or not to break off of the turning
//circle and proceed to the destination.
```

```
bool PathPlanner::poll(float lati, float longi, float beari, bool
turnRight)
{
	//Holds a true value if the breakoff tangent has been found
	bool breakOff1=false;
	float newBearing;
	//Transfer variables for ease of use
	float lat1=lati;
	float lat2=destLat;
	float lon1=longi;
	float lon2=destLon;
	float angleWidth=45.0;

	//Get new bearing to destination based on location around turning
	//circle
	newBearing=getBearing(lat1, lon1, lat2, lon2);

	//If the turn is to the left
	if(!turnRight)
	{
		//If the current heading around the turn is less than or
		//equal to 45 degrees true
		if(beari<=angleWidth)
		{
			//Checks to see if destination is within 45 degrees
			//to the left of the current heading
			if(newBearing<=beari || newBearing >= (360.0-
			(angleWidth-beari)))
				breakOff1=true;
			else
				breakOff1=false;
		}
		//If the current heading around the turn is greater than 45
		//degrees true
		else
		{
			//Checks to see if destination is within 45 degrees
			//to the left of the current heading
			if(newBearing>=beari-angleWidth && newBearing <=
			beari)
				breakOff1=true;
			else
				breakOff1=false;
		}
	}
	//If the turn is to the right
	if(turnRight)
	{
		//If the current heading around the turn is less than 315
		//degrees true
		if(beari<360.0-angleWidth)
		{
			//Checks to see if destination is within 45 degrees
			//to the right of the current heading
			if(newBearing>=beari && newBearing <= beari +
			angleWidth)
				breakOff1=true;
```

```
                  else
                        breakOff1=false;
            }
            //If the current heading around the turn is greater than
            //315 degrees true
            else
            {
                  //Checks to see if destination is within 45 degrees
                  //to the right of the current heading
                  if(newBearing>=beari || newBearing<=
                  ((beari+angleWidth)-360.0))
                        breakOff1=true;
                  else
                        breakOff1=false;
            }
      }
      //Return whether or not to break off of turn circle
      return breakOff1;
}


//Author: Stephen Schall
//Description: Finds a latitude and longitude that are in between
//two given coordinates.  Equation from (Williams, 2004),
//(http://williams.best.vwh.net/avform)
void PathPlanner::intermediatPt(float lat1, float lon1, float lat2,
float lon2, float& intPtLat, float& intPtLon)
{
      //Create variables to be used in calculation
      float A, B, x, y, z, d;
      //Get the distance between the inputted points
      d=measureDistance(lat1, lon1, lat2, lon2);

      //Convert variables to radian distances
      lat1=lat1*3.14159265/180.0;
      lat2=lat2*3.14159265/180.0;
      lon1=-lon1*3.14159265/180.0;
      lon2=-lon2*3.14159265/180.0;
      d=(3.14159265/(180.0*60.0405))*d;

      //Conduct calculation to find median latitude and longitude
      A=sin((1-0.5)*d)/sin(d);
      B=sin(0.5*d)/sin(d);
      x = A*cos(lat1)*cos(lon1) + B*cos(lat2)*cos(lon2);
      y = A*cos(lat1)*sin(lon1) + B*cos(lat2)*sin(lon2);
      z = A*sin(lat1) + B*sin(lat2);
      intPtLat=atan2(z,sqrt((x*x)+(y*y)));
      intPtLon=atan2(y,x);

      //Convert latitude and longitude to degrees
      intPtLat = ((intPtLat*180.0)/3.14159265);
      intPtLon = -((intPtLon*180.0)/3.14159265);

      return;
}
```

```
//Author: Stephen Schall
//Description: PathPlanner basic constructor.
PathPlanner::PathPlanner(void)
{
	return;
}


//Author: Stephen Schall
//Description: Checks whether the UAV could beat the WSN
//contact to a point along the contact's estimated path.
bool PathPlanner::test(float time, float lat2, float lon2)
{
	//Get the distance from the UAV to the point
	float d=measureDistance(curLat, curLon, lat2, lon2);
	//Time it would take the UAV to get to the location
	float timeToIntercept=((d/.000539956803)/maxSpeed);
	//Compare the UAV's time with the contact's time
	if(timeToIntercept<=time)
		return true;
	else
		return false;
}


//Author: Stephen Schall
//Description: Checks whether interception of a contact is possible
//before entering into an exhaustive search to find it.
bool PathPlanner::possible(float lat1, float lon1, float lat2, float
lon2, float contactBear)
{
	//If the contact is moving away from the UAV in X and Y
	bool movingAway;

	//If contact is to the N of the UAV
	if(lon1==lon2 && lat1<lat2)
	{
		//If contact is traveling N
		if(contactBear>=270 || contactBear<=90)
			movingAway=true;
		else
			movingAway=false;
	}
	//If contact is to the NE of the UAV
	else if(lon1<lon2 && lat1<lat2)
	{
		//If contact is traveling NE
		if((contactBear<=90 && contactBear>=0) || contactBear==360)
			movingAway=true;
		else
			movingAway=false;
	}
	//If contact is to the E of the UAV
	else if(lon1<lon2 && lat1==lat2)
	{
		//If contact is traveling E
```

```
	if((contactBear<=180 && contactBear>=0) ||
	contactBear==360)
		movingAway=true;
	else
		movingAway=false;
}
//If contact is to the SE of the UAV
else if(lon1<lon2 && lat1>lat2)
{
	//If contact is traveling SE
	if(contactBear<=180 && contactBear>=90)
		movingAway=true;
	else
		movingAway=false;
}
//If contact is to the S of the UAV
else if(lon1==lon2 && lat1>lat2)
{
	//If contact is traveling S
	if(contactBear<=270 && contactBear>=90)
		movingAway=true;
	else
		movingAway=false;
}
//If contact is to the SW of the UAV
else if(lon1>lon2 && lat1>lat2)
{
	//If contact is traveling SW
	if(contactBear<=270 && contactBear>=180)
		movingAway=true;
	else
		movingAway=false;
}
//If contact is to the W of the UAV
else if(lon1>lon2 && lat1==lat2)
{
	//If contact is traveling W
	if((contactBear<=360 && contactBear>=180) ||
	contactBear==0)
		movingAway=true;
	else
		movingAway=false;
}
//If contact is to the NW of the UAV
else if(lon1>lon2 && lat1<lat2)
{
	//If contact is traveling NW
	if((contactBear<=360 && contactBear>=270) ||
	contactBear==0)
		movingAway=true;
	else
		movingAway=false;
}


//Check if contact is going as fast or faster than the max speed
//of the UAV, and moving away
```

```
	if(maxSpeed<=tarSpeed && movingAway)
	{
		return false;
	}
	//Check if the contact will remain out of range of the UAV,
	//even if UAV is moving faster
	else if(movingAway && (distanceToSN>=maxRange))
	{
		return false;
	}
	else
		return true;
}


//Author: Stephen Schall
//Description: Returns the bearing between two given points (Geographic
//coordinates).  Equation from (Movable Type Scripts, 2006),
//(http://www.movable-type.co.uk/scripts/LatLong.html)
float PathPlanner::getBearing(float lat1, float lon1, float lat2, float
lon2)
{
	//True bearing from point 1 to point 2
	float newBearing;

	//Get distance between the points
	float d=measureDistance(lat2, lon2, lat1, lon1);

	//Convert variables to radians
	d=(3.14159265/(180.0*60.0405))*d;
	lat1=lat1*3.14159265/180.0;
	lat2=lat2*3.14159265/180.0;
	lon1=lon1*3.14159265/180.0;
	lon2=lon2*3.14159265/180.0;

	//Get true bearing
	newBearing=atan2(sin(lon2-lon1)*cos(lat2), cos(lat1)*sin(lat2)-
	sin(lat1)*cos(lat2)*cos(lon2-lon1));

	//Convert bearing to degrees
	newBearing=(180/3.14159265)*newBearing;

	//Convert negative bearings to positive
	if(newBearing<0)
		newBearing=360+newBearing;

	return newBearing;
}


//Author: Stephen Schall
//Description: Determines the amount of time the UAV
//will take to turn before it can proceed on an intercept
//path with a WSN contact
float PathPlanner::addTimeForTurn(float interceptBearing)
{
	//Turn time required
```

```
	float timeAdded=0;
	//Holds a 1 if a turn is required, 0 otherwise
	int turnYesOrNo;
	//Number of steps around turn circle UAV must make,
	//where 8 is a full circle
	int numTurns=0;

	//Get bearing to intercept
	destBearing=interceptBearing;

	//If a turn is required, will return a 1
	turnYesOrNo=pathDecide(1);
	//Check if a turn is required
	if (turnYesOrNo==1)
	{
		numTurns=makeTurn(1);
	}
	//Compute amount of time required for turn
	timeAdded=((3.14159265*turnRadius*2.0)/maxSpeed)*(numTurns/8.0);
	return timeAdded;
}


//Author: Stephen Schall
//Description: Determines if the contact is within a 10 degree sector
//directly in front of the UAV's current bearing.
bool PathPlanner::straightAhead(void)
{
	//Current bearing
	float curHolder;
	//Destination bearing
	float destHolder;

	//Checks whether or not the bearings must be moved for
	//this computation
	if(curBearing>=5.0 && curBearing<=355.0)
	{
		//If destination is within a 5 degrees to the left or right
		//of the UAV's current bearing
		if(curBearing+5.0>=destBearing && curBearing-
		5.0<=destBearing)
			return true;
		else
			return false;
	}
	else
	{
		//Moves current bearing out of contentious range by adding
		//100 degrees
		curHolder=(fmod((curBearing+100.0),360.0)*360.0);
		//Moves destination bearing out of contentious range by
		//adding 100 degrees
		destHolder=(fmod((destBearing+100.0),360.0)*360.0);

		//If destination is within a 5 degrees to the left or right
		//of the UAV's current bearing
		if(curHolder+5.0>=destHolder && curHolder-5.0<=destHolder)
```

```
                return true;
            else
                return false;
        }
        return false;
    }
```

```
////////////////////////////////////////////////////////
//Name: PathPlanner.h
//
//Authors: Stephen Schall
//
//Description: Header file for flight plan construction within
//PathPlanner.cpp.
////////////////////////////////////////////////////////

#pragma once

#include "DevDemo.h"
#include <fstream>
#include <sstream>
#include <math.h>

using namespace std;


// Author: Stephen Schall
// Description: Data structure to hold current UAV data.
struct UAV
{
	//UAV position (degrees)
	float latUAV;
	float lonUAV;
	//UAV current heading (degrees)
	float BearingUAV;
	//UAV speed (m/s)
	float SpeedUAV;
	//UAV altitude (m)
	float altitudeUAV;
};


// Author: Stephen Schall
// Description: Data structure to hold WSN PCPTAv1 input.
struct SNInput
{
	//Position of WSN cluster reporting intrusion (degrees)
	float latNW;
	float lonNW;
	//Contact true heading (degrees)
	float contactBearing;
	//Contact velocity (m/s)
	float contactSpeed;
};


// Author: Stephen Schall
// Description: The class that handles UAV flight plan production
// and two-dimensional path optimization.
class PathPlanner
{
	public:
		//UAV turn radius
		float turnRadius;
```

```
//Assists with tuning maneuverability to different
//platforms
float multiplier;
//UAV current position
float curLat;
float curLon;
//Destination coordinate variables
float destLat;
float destLon;
//Current UAV true bearing
float curBearing;
//True bearing to destination
float destBearing;
//Distance to destination
double distance;
//Distance to instigating WSN cluster
double distanceToSN;
//Tells program whether to follow or to loiter around
//sensor
//network coordinates
bool follow;
//UAV current speed
float uavSpeed;
//WSN contact velocity
float tarSpeed;
//WSN contact true heading
float tarBearing;
//WSN cluster location
float tarLat;
float tarLon;
//Holds WSN PCPTAv1 input text
string contct;
//Holds UAV data
UAV uav1;
//Stream for text parsing
ofstream out;
//UAV's max range in nm
float maxRange;
//UAV's max speed in m/s
float maxSpeed;
//WSN PCPTAv1 input
SNInput in;

//PathPlanner Class function declarations
PathPlanner(string con, bool chase1, float uavIn[], float
mult, float mRange, float mSpeed);
~PathPlanner(void);
string planner();
void makeGotoBeforeTurn();
int makeTurn(int type);
SNInput sensorNwInput(void);
float getInterceptBearing();
double measureDistance(double lat1, double lon1, double
lat2, double lon2);
int pathDecide(int type);
void makeGotoFlyStraight(void);
```

```
            void nextPoint(double lat1, double lon1, double bearing,
            double d, double& lat2, double& lon2);
            bool poll(float lati, float longi, float beari, bool
            turnRight);
            void intermediatPt(float lat1, float lon1, float lat2,
            float lon2, float& intPtLat, float& intPtLon);
            PathPlanner(void);
            bool test(float time, float lat2, float lon2);
            bool possible(float lat1, float lon1, float lat2, float
            lon2, float contactBear);
            float getBearing(float lat1, float lon1, float lat2, float
            lon2);
            float addTimeForTurn(float interceptBearing);
            bool straightAhead(void);
};
```

```
////////////////////////////////////////////////////////////
//Name: SocketConnector.cpp
//
//Author: Procerus Technologies (http://www.procerusuav.com/)
//
//Description: Creates socket connections from PCPTAv1 to the
//VCDI, which passes command packets to the Kestrel Autopilot
//via VC
////////////////////////////////////////////////////////////////
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "SocketConnector.h"

using namespace std;
CSocketConnector::CSocketConnector(const CString ipAddress, HWND
MainHwnd)
{
	//All we want to do is connect to the Dev Server created by
	//Virtual Cockpit
	m_VCServerConnected = FALSE;
	m_DataThreadRunning = FALSE;
	m_VCSocket = INVALID_SOCKET;
	m_MainHwnd = MainHwnd;

	//Try to connect to the VC
	if (WSAStartup( MAKEWORD(1,1), &m_wsaData ) == NO_ERROR)
	{
		m_VCSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

		if (m_VCSocket != INVALID_SOCKET)
		{
			// Connect to the data server.
			clientService.sin_family = AF_INET;
			clientService.sin_addr.s_addr = inet_addr(ipAddress);
			//default VC listening port number
			clientService.sin_port = htons(5005);
			if (connect(m_VCSocket, (SOCKADDR*) &clientService,
			sizeof(clientService)) != SOCKET_ERROR)
			{
				m_VCServerConnected = TRUE;
				AfxBeginThread(StartReadThread, this);
				m_DataThreadRunning = TRUE;
			}
		}
	}
}


CSocketConnector::~CSocketConnector()
{
	m_VCServerConnected = FALSE;
	//AfxMessageBox("Made False 4");
```

```
	//Close the socket
	if (m_VCSocket != INVALID_SOCKET)
		closesocket(m_VCSocket);

	//Wait Until the ReadThread stops
	while (m_DataThreadRunning)
		Sleep(50);

	//Clear out the Comm packets
	while(NumVCPackets() > 0)
		PopVCPacket();

	//Clean up the driver
	WSACleanup();
}


UINT CSocketConnector::StartReadThread(LPVOID pParam)
{
	CSocketConnector *pConnector = (CSocketConnector *)pParam;

	while (pConnector->IsConnected())
		pConnector->ReadData();

	pConnector->m_DataThreadRunning = FALSE;

	return 0;
}


void CSocketConnector::ReadData()
{
	//We should never really go over 1K in data
	unsigned char ReadBuff[1024];

	int BytesRecv = recv(m_VCSocket, (char *)ReadBuff, 1024, 0);

	if ((BytesRecv == 0) || (BytesRecv == SOCKET_ERROR))
	{
		m_VCServerConnected = FALSE;
	}
	else
	{
		//Parse the data
		sVCPacket RecvPkt;

		//Copy it over
		memcpy(&RecvPkt, ReadBuff,BytesRecv);

		//Push it onto the deque
		m_PktContainer.push_back(RecvPkt);

		//Send notification to main app
		::PostMessage(m_MainHwnd, WM_DATA_MSG,0,0);
	}
}
```

```
BOOL CSocketConnector::SendData(sVCPacket *Pkt)
{
	if(m_VCServerConnected)
	{
		if(send(m_VCSocket, (const char *)Pkt, Pkt-
>DataSize+2*sizeof(int), 0) != SOCKET_ERROR)
			return TRUE;
	}

	return FALSE;
}


sVCPacket CSocketConnector::GetNextVCPacket()
{
	//Send the front of the list
	sVCPacket Pkt = m_PktContainer.front();
	return Pkt;
}


void CSocketConnector::PopVCPacket()
{
	m_PktContainer.pop_front();
}
```

```
////////////////////////////////////////////////////////////
//Name: SocketConnector.h
//
//Author: Procerus Technologies (http://www.procerusuav.com/)
//
//Description: Header file for creating socket connections from
//PCPTAv1 to the VCDI, which passes command packets to the
//Kestrel Autopilot via VC.
////////////////////////////////////////////////////////////////
// (C) 2005 Procerus Technologies, all rights reserved.
// It is unlawful to use this source code except by
// license from Procerus Technologies as part of the
// Virtual Cockpit and Kestrel Autopilot System.
////////////////////////////////////////////////////////////////

#ifndef SOCKET_CONNECTOR_H
#define SOCKET_CONNECTOR_H

#include <deque>

//Message to notify main window we received data
#define WM_DATA_MSG                 WM_USER + 1055


//Packet types defined in VC interface
//This packet when received by the VC will pass all data straight to
//the autopilots
#define VC_PASSTHROUGH              10
//This packet setups the VC to forward messages it receives from the
//autopilots
#define VC_FRWD_PKT_SETUP           20

struct sVCPacket
{
      //Packet interface type to the VC Development server
      int VCPacketType;
      //The size of the data in the char array
      int DataSize;
      //The data associated with this packet
      unsigned char PktData[1024];
};

struct sPassThrough
{
      //The destination address of the pass through packet...usually
      //airplane address
      unsigned short DestAddr;
      //The data that makes up the pass through packet...refer to
      //Kestrel Communications Guide
      unsigned char PassData[128];  };

class CSocketConnector
{
      public:
            CSocketConnector(const CString ipAdress, HWND MainHwnd);
            ~CSocketConnector();
```

```
            //Returns if it is still connected to the Slope Soaring
            //Simulator
            BOOL IsConnected() const { return m_VCServerConnected; };

            //Read Data from the Socket
            void ReadData();

            //Sends the data and returns if it was successful or not
            BOOL SendData(sVCPacket *Pkt);

            //Returns a single packet
            sVCPacket GetNextVCPacket();

            //Removes the parsed comm packet
            void PopVCPacket();

            //Returns the number of packets in queue
            int NumVCPackets(){ if(m_PktContainer.empty()) return 0;
            return (int)m_PktContainer.size(); }

      protected:
            //Runs the ReadThread and continues reading until
            //disconnected
            static UINT StartReadThread(LPVOID pParam);

      protected:
            //Used to load the Driver for the socket
            WSADATA m_wsaData;
            //The Socket to send/reveieve the data to the VC
            SOCKET m_VCSocket;
            //Socket to the server
            sockaddr_in clientService;

            //If a connect is established
            BOOL volatile m_VCServerConnected;
            //If the Read Thread is still running
            BOOL volatile m_DataThreadRunning;
            //Holds the packets that haven't been read by the dev app
            std::deque<sVCPacket> m_PktContainer;
            //Main window handle for sending messages
            HWND m_MainHwnd;
};

#endif
```

LIST OF REFERENCES

Al-Karaki, Kamal. "A Taxonomy of Routing Techniques in Wireless Sensor Networks." In Ilayas, Mahgoub (Eds) Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems. Boca Raton.

Allen, Quinn, Bachmann, and Ritzmann. "Abstracted Biological Principles Applied with Reduced Actuation Improve Mobility of Legged Vehicles." Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '03), vol. 2, pp. 1370-1375, 2003, Las Vegas, USA.

"Bearing Between Two Points." The Math Forum. 20 Dec. 2001. Drexel University. 10 Jun 2006 <http://mathforum.org/library/drmath/view/55417.html>.

Bharathidasan, Archana, and Vijay Anand Sai Ponduru, comps. Sensor Networks: an Overview. University of California, Davis. 9 Jun 2006 <http://wwwcsif.cs.ucdavis.edu/~bharathi/sensor/survey.pdf>.

Boria, Frank, Richard Bachmann, Peter Ifju, Roger Quinn, Ravi Vaidyanathan, Chris Perry, and Jeffrey Wagener, comps. A Sensor Platform Capable of Aerial and Terrestrial Locomotion. University of Florida, BioRobots, LLC, Case Western Reserve University, Naval Postgraduate School. 19 Apr 2006 <http://www.nps.navy.mil/se/ravi/My%20Papers/MMALV%20IROS%202005.pdf>.

"Calculate Distance and Bearing Between Two Latitude/Longitude Points." Movable Type Scripts. Movable Type Ltd. 21 Apr 2006 <http://www.movable-type.co.uk/scripts/LatLong.html>.

Culler, David, Deborah Estrin, and Mani Srivastava. "Overview of Sensor Networks." IEEE Computer Society Aug. 2004. 7 Jun 2006 <http://www.ics.uci.edu/~dutt/ics212-wq05/ieeecomput-sensornet-aug04.pdf>.

Egli, Peter. Susceptibility of Wireless Devices to Denial of Service Attacks. 2006. NetModule AG. 23 May 2006 <http://www.netmodule.com/store/publications/susceptibility_of_wireless_devices_to_DoS.pdf>.

Estrin, Deborah, Mani Srivastava, and Akbar Sayeed. Wireless Sensor Networks. MobiCom 2002 Tutorial T5: the Eighth ACM International Conference on Mobile Computing and Networking, 23 Sep 2002, Association of Computing Machinery. 26 June 2006 <http://nesl.ee.ucla.edu/tutorials/mobicom02/>.

"Haversine Formula." Wikipedia. 8 May 2006. Wikimedia Foundation, Inc. 10 May 2006. <http://en.wikipedia.org/wiki/Haversine_formula>.

Hill, Jason, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, comps. System Architecture Directions for Networked Sensors. Department of Electrical Engineering and Computer Sciences, University of Califronia, Berkley. 22 Apr 2006 <http://www.tinyos.net/papers/tos.pdf>.

Ifju, Ettinger, Jenkins, Lian, Shyy and Waszak. "Flexible-Wing-Based Micro Air Vehicles." 40th AIAA Aerospace Sciences Meeting, January 2002.

Intanagonwiwat, Govindan, Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000), August 2000.

"Kestrel Autopilot V2.22." 31 Mar. 2006. Procerus Technologies. 10 Jun 2006 <http://www.procerusuav.com/Documents/Kestrel_2.22.pdf>.

Kestrel Autopilot System: Communications Protocol 2.0 for Virtual Cockpit 2.1 and Kestrel Autopilot MA6 Level Firmware. Procerus Technologies, 2006.

Kulik, Heinzelman, Balakrishnan. "Negotiation-based Protocols for Disseminating Information in Wireless Sensor Networks." *Wireless Networks*. March 2002.

Lambrecht, A. D. Horchler, and R. D. Quinn. "A Small Insect Inspired Robot that Runs and Jumps." Proceeding of the IEEE International Conference on Robotics and Automation (ICRA '05), Barcelona, Spain, 2005.

Lewis, F. L. "Wireless Sensor Networks." Smart Environments: Technologies, Protocols, and Applications. 2004.

"Mission Statement." TinyOS. University of California, Berkeley. 10 Jun 2006 <http://www.tinyos.net/special/mission>.

"MoteView Users Manual." Crossbow. June 2006. Crossbow Technology, Inc. 10 Jun 2006<http://www.xbow.com/Support/Support_pdf_files/MoteView_Users_Manual.pdf >.

Mueller, T J. "The Influence of Laminar Separation and Transition on Low Reynold's Number Airfoil Hysteresis." J. Aircraft 22 (1985): 763.

".NET Framework." Wikipedia. 4 July 2006. Wikimedia Foundation, Inc. 5 Jul 2006 <http://en.wikipedia.org/wiki/.NET_Framework>.

"Readme: Microsoft Visual Studio 2005 Setup Issues." Microsoft Visual Studio Developer Center. Microsoft Corporation. 20 Jun 2006 <http://msdn.microsoft.com/vstudio/support/readme>.

Salatas, Vlasios. "Object Tracking Using Wireless Sensor Networks." Naval Postgraduate School, Thesis, 2005.

"Servomechanism." Wikipedia. 28 Jun 2006. Wikimedia Foundation, Inc. 29 Jun 2006 <http://en.wikipedia.org/wiki/Servomechanism>.

Shah, Rabaey. Energy Aware Routing for Low Energy Ad Hoc Sensor Networks. IEEE Wireless Communications and Networking Conference (WCNC*)*, 17 Mar 2002, Institute of Electrical and Electronics Engineers.

Sinnott, R W. "Virtues of the Haversine." Sky and Telescope 68 (1984): 159.

"TinyOS." Wikipedia. 23 Apr. 2006. Wikimedia Foundation, Inc. 25 Apr 2006 <http://en.wikipedia.org/wiki/TinyOS>.

"UAV Flight Guide: Version 1.1." Kestrel Autopilot System. 20 Jun 06. Procerus Technologies. 28 Jun 2006 <http://www.procerusuav.com/Documents/UAV_Flight_Guide.pdf>.

"UAV Test Platform: RC Hobby --EPP Foam Wing." Procerus Technologies. 10 Jun 2006 <http://www.procerusuav.com/Documents/UAV_platform.pdf>.

User Guide: Virtual Cockpit 2.1, Kestrel Autopilot (Firmware Version MA6). Procerus Technologies. 2006.

Williams, Ed. "Aviation Formulary V1.42." Ed Williams' Aviation Page. 5 Jul 2004. <http://williams.best.vwh.net/avform.htm>.

"Winglet." Wikipedia. 27 June 2006. Wikimedia Foundation, Inc. 2 Jul 2006 <http://en.wikipedia.org/wiki/Winglet>.

"Wireless Sensor Network." Wikipedia. 12 May 2006. Wikimedia Foundation, Inc. 13 May 2006 <http://en.wikipedia.org/wiki/Wireless_sensor_network>.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Gurminder Singh
   Naval Postgraduate School
   Monterey, CA

4. Ravi Vaidyanathan
   Naval Postgraduate School
   Monterey, CA

5. Robert Bledsoe
   United States Marine Corps
   Monterey, CA

6. Joshua Kiihne
   United States Marine Corps
   Monterey, CA

8. Stephen Schall
   United States Navy
   Charleston, SC